ADDALY · THE AI CLASSROOM

The Open Model Ecosystem

Llama, Mistral, Qwen, DeepSeek, Gemma, Phi: what is real and what is marketing.

7 modules · 71 lessons · 28 figures · 7 case studies · 49,110 words · about 10 hours

Dr Mustafa Mun
Author and editor

ABOUT THIS BOOK

The Open Model Ecosystem, 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/open-models. 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. What open actually means

Given any model release, say precisely which freedoms it grants and which it withholds, trace its licence through every fine-tune in the chain, verify the files you downloaded are the files the publisher made, and state what an auditor or a regulator could still ask that the weights alone cannot answer

1. Open Weights Is Not Open Source
2. Reading the Licence Before You Ship
3. Why a lab gives away a model
4. Grading openness without arguing about the word
5. Where the training data comes from
6. The copyright question nobody has settled
7. What regulators ask of an open model
8. Trusting a file you did not build
9. Vetting a stranger's fine-tune
10. Publishing weights people can trust
11. Case study: The MIT badge that was not theirs
12. Module test

2. The map, and how to read it

Place any open release on a map of families, modalities, formats and serving options within ten minutes — naming its licence posture, its likely tokenizer, the tools that will serve it and the size class it competes in — and pick the right kind of model for a retrieval, code, speech or generation task rather than reaching for a chat model by reflex

1. The Families, and What Each Is Actually For
2. Hugging Face as Infrastructure
3. Regional and specialist models
4. Models that write code
5. Embeddings you can host yourself
6. Rerankers, and why retrieval is two stages
7. Speech, in both directions
8. Image and video weights, and their licence traps
9. What a model of each size can honestly do
10. Renting open weights by the token
11. Case study: The best model for Tamil, and the licence on it
12. Module test

3. Inside the files

Open a model repository you have never seen and predict, from config.json and the tokenizer alone, its parameter count, its memory footprint at a given context length, its per-token cost in your languages and the exact conversation format it requires — then verify each prediction with a command and diagnose a runaway or garbled generation from the files rather than by guessing

1. Base, Instruct, Reasoning
2. Every number in config.json
3. Why the same sentence costs different amounts
4. The chat template is a contract
5. Attention variants, and the cache that eats your memory
6. What a 128k context window really means
7. Mixture of experts, and the memory it still needs
8. Models that can see
9. Formats, and what conversion costs you
10. Why it will not stop talking
11. Case study: The model that would not stop talking, two days before the batch
12. Module test

4. Making it run on the machine you have

Size, quantise and serve an open model on hardware you actually have — a laptop, a phone, a free cloud tier or a rented GPU — predict its generation speed from memory bandwidth before running it, choose a bit width you can defend, and state the monthly token volume at which renting stops being cheaper than owning

1. Choosing a Model for the Job
2. What quantisation actually does
3. Choosing a bit width you can defend
4. llama.cpp, end to end
5. Ollama and the friendly path
6. Predicting speed before you download
7. Serving many users at once
8. Running all of this without money
9. Rent, or own: the arithmetic
10. What actually runs on a phone
11. Case study: Eleven branches, one radiologist, and data that cannot leave
12. Module test

5. Getting good output

Take an open model producing bad output and identify whether the fault lies in the sampler, the prompt, the examples, the schema, the language or the model itself — then apply the cheapest correct fix, from a temperature change to a grammar-constrained decoder to a separate guardrail model, and demonstrate the improvement on a held set

1. The sampler, and the defaults that ship broken
2. Making the output valid by construction
3. What changes when nothing is hidden
4. Examples, and how to choose them
5. Tool calling, and what breaks at 8B
6. Agent loops on models that are not clever
7. What these models really do in your language
8. Refusals, and the uncensored question
9. Guardrails as a separate layer
10. When the model does not know
11. Case study: The classifier that refused the messages it existed for
12. Module test

6. Adapting a model to your work

Decide with evidence whether a task needs fine-tuning at all; if it does, build the dataset, size a LoRA to fit free hardware, run it, and demonstrate on a held-out set and a regression suite that the adapted model is better at the task and no worse at everything else

1. When not to fine-tune
2. What a LoRA actually is
3. Fitting a fine-tune in 16 GB
4. Five hundred examples, done properly
5. A real run, on free hardware
6. Teaching it which answer is better
7. Distilling a larger model into a smaller one
8. Measuring what you broke
9. Merging models, and what it really does
10. When the model does not have your language at all
11. Shipping the adapter
12. Case study: Nine points better, and worse at everything else
13. Module test

7. Judging models and claims

Settle a model choice with evidence you produced yourself — a private held-out set, a pinned harness, a quantisation compared against its full-precision reference, latency measured at real concurrency and cost expressed per task rather than per token — and explain to a sceptical colleague which published claims about that model you believe and why

1. What Benchmarks Actually Measure
2. Reading a model card for what it does not say
3. The leaderboards that still carry information
4. Running the standard harness yourself
5. Proving what a quantisation cost
6. Evaluating an embedding model on your corpus
7. Measuring latency properly
8. Cost per task, not cost per token
9. Swapping the model under a live product
10. Evaluate a Model on Your Task in an Afternoon
11. Case study: The leaderboard model and the committee
12. Module test

The Open Model Ecosystem — final examination

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

MODULE 1

What open actually means

The word open is doing a great deal of work in this ecosystem, and it means something different in every sentence it appears in. This block separates the freedoms a release actually grants — run it, adapt it, redistribute it, inspect it, rebuild it — from the ones the announcement implies, and gives you the licence chain, the data question, the regulatory position and the supply-chain checks that a model you are about to depend on has to survive.

> **BY THE END OF THIS MODULE YOU CAN**
>
> Given any model release, say precisely which freedoms it grants and which it withholds, trace its licence through every fine-tune in the chain, verify the files you downloaded are the files the publisher made, and state what an auditor or a regulator could still ask that the weights alone cannot answer

1 · 7 min

Open Weights Is Not Open Source

THE ONE THING TO KEEP

Open weights means you can run and adapt the model, not that you can see how it was made.

What actually lands on your disk

When a lab "open sources" a model, you get a folder. Look inside a typical one:

```
config.json                # layer count, hidden size, vocab
size
tokenizer.json             # how text becomes numbers
model-00001-of-00004.safetensors
model-00002-of-00004.safetensors
...
README.md                  # the model card
LICENSE
```

That is the model. Billions of floating point numbers, plus enough metadata to load them.

Here is what is not in the folder: the training data. The code that filtered and mixed that data. The training code. The intermediate checkpoints. The human preference data used for the final tuning. The failed runs. In almost every case, none of that ships.

The analogy, and where it breaks

Weights are closer to a compiled binary than to source code. You can run it. You can redistribute it. You cannot rebuild it, and you cannot read what went in.

The analogy breaks in one useful place. You can meaningfully modify a binary blob of weights by fine-tuning it, which is cheap and works. So the practical freedom is larger than "here is a .exe" suggests. It is still not the freedom to reproduce.

Why this matters when you are not being pedantic

Four things you cannot do with weights alone:

- **Audit the data.** You cannot check whether your competitor's private documents, a copyrighted corpus, or the benchmark you are about to run are in there.
- **Reproduce.** You cannot rebuild the model with one thing changed. Nobody outside the lab can.
- **Remove something.** If the model learned a fact you need gone, you can suppress it at inference or fine-tune against it. You cannot delete it from the source.
- **Explain a behaviour by pointing at data.** Every explanation is behavioural, from the outside.

One thing you can do, which is the durable benefit: keep it. If the lab shuts the API down, deprecates the model, changes the price, or decides your use case is off-policy, your copy on disk still runs. That is the real reason open weights matter, and it is enough.

The definition people argue about

The Open Source Initiative published an Open Source AI Definition in October 2024. It asks for three things: the parameters, the complete training and inference code, and detailed information about the data — enough that a skilled person could build a substantially equivalent system. Almost no popular model meets it, because almost none release the data information.

So three labels are worth keeping separate:

- **Open weights.** You get the numbers. Terms vary. This is most of the ecosystem: Llama, Gemma, and much else.

- **Openly licensed weights.** You get the numbers under Apache 2.0 or MIT, with no use restrictions. Mistral's main line, Qwen3, DeepSeek-R1, Phi-4, IBM Granite. Still no data.
- **Fully open.** Weights, data, training code, checkpoints, evaluation. AI2's OLMo, EleutherAI's Pythia, Hugging Face's SmolLM. A small group, and the only one where "open source" is honest without qualification.

Most of the loud releases are in the first two buckets. That is not a scandal. It is just not what the word implies to someone who has used Linux.

Three questions to ask

When somebody tells you a model is open, ask:

1. Can I run it offline, on my own hardware, forever? (Almost always yes.)
2. Can I use it commercially without asking anyone? (Usually, with conditions. Next lesson.)
3. Can I see what went into it? (Almost never.)

If you say "open weights" instead of "open source", you have said the true thing and you have lost nothing. Use the accurate word and you will make better decisions downstream, because the accurate word tells you which questions remain open.

BEFORE YOU MOVE ON

A team plans to adopt an open-weights model in a hiring tool, arguing that being open means they can audit it for bias before deployment. What is the flaw in that argument?

A. Open weights let them measure the model's behaviour on test cases, but not inspect the training data, so the audit can only be behavioural.

B. The model's licence forbids auditing, so the plan is not permitted.

C. There is no flaw, because the weights are a compressed encoding of the training data and can be decoded back into it.

D. The model must meet the OSI Open Source AI Definition before an audit of it would be legally valid.

2 · 8 min

Reading the Licence Before You Ship

THE ONE THING TO KEEP

A licence travels with every fine-tune and every download; check the chain, not the badge on the page.

I am not a lawyer and this is not legal advice. It is a list of the things that surprise people, so you know what to hand to someone who is.

Three families

Genuine open source. Apache 2.0 or MIT. Mistral's main line, Qwen3, DeepSeek-R1, Phi-4, OLMo, IBM Granite, SmolLM. Keep the notice, do what you like. No user caps, no acceptable use policy, no naming rules. If two candidates are close and one is Apache 2.0, that is a real tiebreaker.

Bespoke community licences. Llama, Gemma, and some Qwen releases. Free for nearly everyone, with conditions attached. Written by the lab, not by lawyers who expected you to redistribute.

Non-commercial or non-production. Cohere's Command R family under CC-BY-NC. Mistral's Codestral under a non-production licence. A large share of research fine-tunes on Hugging Face. You can prototype. You cannot ship.

The clauses people miss

User thresholds. Llama's licence stops being free above 700 million monthly active users, measured before the model's release date. Some Qwen releases use 100 million. This will not bite you. It bites the company that acquires you, and it is the first thing their diligence checks.

Naming and attribution. Llama requires "Built with Llama" displayed prominently, a specific attribution line in your notice file, and any model you

derive from it must have a name that *starts with* "Llama". Teams discover this after printing the branding.

Acceptable use policies incorporated by reference. Gemma's terms attach a prohibited use policy that Google can update, and reserve Google's right to restrict use it believes violates that policy. You are agreeing to a document that can change after you ship.

Territory. Llama 4's licence carried a restriction on the multimodal models for entities domiciled in the EU. Read the geography clause if you have European users or a European entity.

What you may do with outputs. Some licences restrict using generations to train other models. Llama's recent terms allow it if the resulting model's name begins with "Llama". If your plan is to generate synthetic training data, this clause is the whole plan.

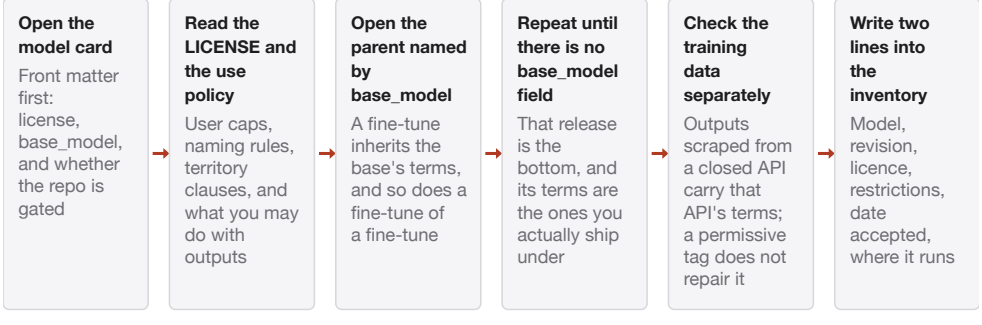
The inheritance trap

This is the one that catches careful people.

`DeepSeek-R1-Distill-Llama-70B` is a Llama model that was trained on R1's reasoning traces. DeepSeek-R1 itself is MIT. The distill is not. It carries Meta's community licence, because the base weights are Meta's.

A fine-tune inherits the base model's licence. So does a fine-tune of a fine-tune. Follow the chain to the bottom:

Tracing a licence to the bottom of the chain



DeepSeek-R1 is MIT. DeepSeek-R1-Distill-Llama-70B is not, because the base weights are Meta's. The badge on the page is the first link in the chain, never the answer.

```
# the front matter of a model card names its parent
hf download deepseek-ai/DeepSeek-R1-Distill-Llama-70B README.md
--local-dir ./card
head -20 ./card/README.md    # look for: license:, base_model:
```

Repeat on the parent. Keep going until you reach something with no `base_model` field.

And separately from the weights: the *data* a fine-tune was trained on has its own terms. A great many popular community fine-tunes were trained on outputs scraped from a closed commercial API, which typically breaches that API's terms of service. An Apache 2.0 badge on the resulting weights does not repair that.

A five minute check

For every model you are seriously considering:

```
hf download <org>/<model> LICENSE USE_POLICY.md README.md --lo-
cal-dir ./chk
```

Read the LICENSE. Read the use policy. Read the README front matter for `license`, `base_model`, and whether the repo is gated.

On gated repos: clicking "agree" on Hugging Face is you entering an agreement. A CI job with a shared token that inherits your acceptance is not a separate acceptance, and mirroring the weights to a public bucket to dodge the gate breaks the terms you agreed to.

Write it down once

Keep a two-line-per-model inventory in the repo:

```
Qwen/Qwen3-8B | rev 9a3f21c | Apache-2.0 | no restrictions |
prod: summarizer
google/gemma-3-4b-it | rev 4b1e07d | Gemma Terms | prohibited-
use policy, updatable | prod: on-device
```

It takes ten minutes to start and saves a bad week later, when someone asks what you are shipping and the honest answer has to be assembled from memory.

BEFORE YOU MOVE ON

You fine-tune DeepSeek-R1-Distill-Llama-8B on your own support tickets and want to release the result publicly as "Acme-Assist". Which statement is correct?

A. The distill is built on Llama weights, so Meta's community licence and its naming rules apply to your release; R1's MIT licence does not cover it.

B. DeepSeek-R1 is MIT licensed, so anything distilled from it inherits MIT and you can release freely.

C. Because you fine-tuned it on your own data, the resulting weights are a new original work under whatever licence you choose.

D. MIT and community licences are compatible, so you can pick whichever of the two suits your release best.

---

3 · 9 min

Why a lab gives away a model

THE ONE THING TO KEEP

A lab releases weights when a cheap, ubiquitous model helps something else it sells, so openness tracks strategy rather than principle and can be withdrawn from the next version at any time.

The question worth asking first

Training a model at the top of the open ecosystem costs somewhere between a few million and a few hundred million dollars in compute alone, before salaries. Then the lab publishes the result for anyone to download. If you can-

not explain why, you will be repeatedly surprised by what happens next: a licence that tightens, a family that goes quiet, a flagship that stays behind an API while only the small siblings ship.

The releases are not charity and they are not an accident. Each family has a business reason, and the reason predicts the behaviour.

Commoditise the thing next to what you sell

The oldest strategy in software is to give away the complement of your product. If you sell electricity, cheap kettles help you. Meta does not sell model access; it sells attention and advertising, and it needs models to be cheap, ubiquitous and not controlled by Google or OpenAI. Releasing Llama makes the model layer a commodity and keeps the layer above it — the products — competitive. That is a coherent commercial decision, and it explains why Meta could release a strong model for years without ever charging for one.

Alibaba's Qwen releases sit next to a cloud business. A developer who builds on Qwen and then needs more machines than a laptop has an obvious place to rent them. Google's Gemma works the same way, with the additional defensive purpose of keeping developers inside a Google-shaped toolchain rather than a Meta-shaped one.

Reputation, recruitment and standard-setting

DeepSeek published unusually detailed papers alongside its weights. The direct return on that is not licence revenue; it is being taken seriously, hiring the people who read the papers, and setting the reference implementation that everyone else compares against. Mistral's early Apache releases did the same job for a young European company that needed to be believed in before it could sell anything.

Microsoft's Phi line is closer to a research demonstration: the argument is that carefully built data beats raw scale at small sizes, and the model is the evidence. AI2's OLMo and EleutherAI's Pythia are public-good science, funded to be studied rather than sold.

What this predicts

Three things follow, and each has bitten someone.

Openness tracks distance from the frontier. Labs release what is no longer their crown jewel, or what is strategically useful to commoditise. Where a model is the product, it stays behind an API. Expect the newest and largest thing a commercial lab has to be the least open thing it has.

Terms follow strategy, and strategy changes. Mistral shipped its main line under Apache 2.0 and then shipped Codestral under a licence that forbids production use. Llama 4's terms carried a restriction aimed at entities domiciled in the EU. Stability changed its licensing more than once. None of this was dishonest; the strategy moved and the licence moved with it.

The next version is not promised to you. There is no obligation on any lab to release the successor of a model you built on, under the same terms or at all.

The practical consequences

Do three things and this stops being a risk.

- **Keep the weights you depend on.** A copy on your own storage is the whole point of open weights. A model that is withdrawn from the hub still runs from your disk. Budget the storage; a 16 GB archive is cheaper than a migration.
- **Record the terms you accepted, with the date.** Licences and acceptable-use policies get edited. What binds you is generally the version you accepted, and you cannot prove that later without a copy.
- **Do not design around a single family.** If swapping from a Qwen to a Llama of the same size means rewriting your prompts, your chat template handling and your serving stack, you have built the lock-in that open weights were supposed to prevent.

> *Renting from a hosted provider is not the opposite of this. The freedom that matters is that you could run the weights yourself and nobody can*

> *take the model away from you. Whether you exercise that freedom today is an operational question, not a philosophical one.*

The one genuinely fragile case

Watch for models released by a company whose only product is that model. The economics have to close somewhere. When it does not, the weights that were free become a paid tier, the licence changes at the next release, or the company is acquired and the new owner has different lawyers. This has happened often enough to be a pattern rather than a warning.

BEFORE YOU MOVE ON

A company with no product other than its model releases strong open weights under Apache 2.0. What is the most useful inference to draw about your risk?

A. The Apache licence is irrevocable for the version released, so keep your own copy of those weights and do not assume the successor arrives on the same terms.

B. Apache 2.0 cannot be changed, so the whole family is safe to build on indefinitely.

C. The release is unsustainable, so the model is probably weak and should be avoided.

D. Because the licence is permissive, the training data must also be documented and legally clear.

4 · 9 min

Grading openness without arguing about the word

THE ONE THING TO KEEP

Score a release on six separate axes — weights, licence, data, code, evaluation, checkpoints — because each axis buys a different concrete capability, and only some of them matter for any given job.

Stop arguing, start scoring

Arguments about whether a model is really open source go nowhere because the two sides are talking about different freedoms. The argument ends the moment you replace the word with a short list of axes and score each one. Then a disagreement becomes a specific claim: you needed axis four, and this release scores zero on it.

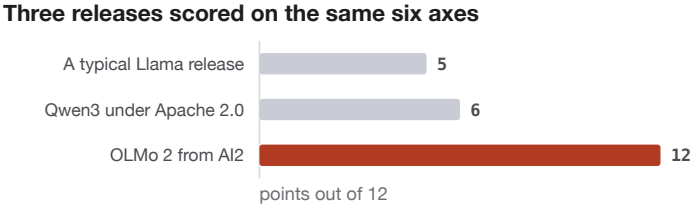
Six axes cover almost everything anybody actually needs.

1. **Weights.** Are the parameters downloadable, and without asking permission? Score 2 for an open download, 1 for a gated repo that requires accepting terms, 0 for API only.
2. **Licence.** Score 2 for an OSI-approved licence with no use restrictions (Apache 2.0, MIT), 1 for a bespoke community licence with conditions, 0 for non-commercial or research-only.
3. **Training data.** 2 if the corpus itself is released, 1 if it is described in enough detail to be checked, 0 if the card says a mixture of publicly available data.
4. **Training code and recipe.** 2 for the actual pipeline, 1 for a paper with hyperparameters, 0 for neither.
5. **Evaluation.** 2 for the harness, the exact commit and the raw per-task outputs, 1 for a table of numbers with the harness named, 0 for a marketing chart.

6. **Checkpoints.** 2 for intermediate checkpoints across training, 1 for the final base as well as the instruct model, 0 for the instruct model only.

What three real releases look like

Roughly, and this moves:



Weights, licence, data, code, evaluation and checkpoints, two points each. The licence axis is the whole difference between the first two, and for a commercial team it is usually the axis that decides. The release scoring twelve is used least, because at any given moment it is not the strongest model at its size.

- **A typical Llama release.** Weights 2, licence 1, data 0, code 0, evaluation 1, checkpoints 1. Total 5 of 12. You can build a business on it. You cannot audit it.
- **Qwen3 under Apache 2.0.** Weights 2, licence 2, data 0, code 0, evaluation 1, checkpoints 1. Total 6. The licence axis is the whole difference, and for a commercial team that axis is often the one that decides.
- **OLMo 2 from AI2.** Weights 2, licence 2, data 2, code 2, evaluation 2, checkpoints 2. Total 12. Fewer people use it, because at any moment it is not the strongest model at its size. That trade is the actual shape of the ecosystem.

Each axis buys a specific thing

This is the part that makes the rubric useful rather than pedantic.

Data disclosure buys contamination checking. If you want to know whether the benchmark you are quoting was in the training set, there is exactly one way to find out, and it is to search the corpus. Without the corpus you are guessing. It also buys copyright diligence: you can answer what is in it rather than reporting that nobody told you.

Training code buys reproduction and, more practically, continued pretraining that does not fight the original recipe. If you plan to train further on your own data, knowing the exact optimiser, schedule and data mix means your run resembles the original rather than knocking it sideways.

Checkpoints buy research on training dynamics, and they buy the option of restarting from an earlier point when the final instruct tuning has removed a behaviour you needed. A base model released alongside the instruct model is the single most useful item on this axis for ordinary work.

Evaluation artefacts buy the ability to reproduce a claimed number. A table of scores with no harness commit is not reproducible even in principle, because the harness moves.

How to use it in ten minutes

Open the model card and the repository file list. Score the six axes. Then ask the only question that matters: which axis does my situation require?

- Shipping a commercial product: axis 2 dominates, and axes 3 to 6 can be zero without hurting you.
- Answering a regulator or a customer security review: axes 3 and 5 become necessary.
- Doing research you intend to publish: axes 4 and 6 or you cannot say anything defensible about causes.
- Fine-tuning for a new language: axis 6, because you want the base rather than the instruct model, and axis 3, because you want to know how much of that language it has already seen.

> *Openwashing is the term for a release that scores 2 on weights, advertises itself as open source, and scores 0 on everything else. The rubric is the answer to it: you do not have to accuse anyone of anything, you just publish the six numbers.*

Write your scores into your model inventory next to the licence. When somebody asks in a year why you picked this model, the six numbers are the an-

swer, and they still make sense after the leaderboard has turned over four times.

BEFORE YOU MOVE ON

An auditor asks you to demonstrate that the benchmark you quote for your deployed model was not present in its training data. Which axis of openness does that require, and what happens if the release scores zero on it?

A. The licence axis; without a permissive licence you have no right to inspect the model's behaviour on the benchmark.

B. The evaluation axis; publishing the harness commit would settle the contamination question directly.

C. The training data axis; with no corpus to search, contamination can only be estimated behaviourally, for example by testing on a freshly written equivalent set.

D. The checkpoints axis; comparing intermediate checkpoints would show when the benchmark was memorised.

---

5 · 10 min

Where the training data comes from

THE ONE THING TO KEEP

Open models are built on a handful of named public web corpora processed by undisclosed filters, and crawler opt-outs only ever work forwards, so what a model already learned cannot be withdrawn.

The corpora that are actually used

Almost every open model of the last few years is built on a small number of public web corpora, mixed with code, books, papers and synthetic text.

Knowing the names is worth an afternoon because the model card usually names them, and if it does not, that absence is itself information.

Common Crawl is a non-profit that has been crawling the web since 2008 and publishes monthly snapshots. Each snapshot is a few billion pages; the archive as a whole runs to hundreds of terabytes of compressed HTML. It is the raw material under nearly everything, and it is raw: boilerplate, navigation menus, spam, duplicated pages, and text in every language mixed together.

C4 is Google's cleaned English extract of a single Common Crawl snapshot, around 750 GB of text. It became the reference example of what filtering does, partly because researchers later audited it and found what the filters had removed — including a disproportionate amount of text written in African American English and text about LGBT topics, because a naive bad-word filter treats reclaimed language as profanity.

The Pile was EleutherAI's 825 GB mixed corpus: code, PubMed, arXiv, Stack Exchange, and a books collection called Books3. Books3 turned out to be roughly 190,000 pirated books, and it was removed after takedown notices in 2023. Models trained before that still contain what it taught them.

Dolma is AI2's 3-trillion-token corpus, released with its tooling. **FineWeb** is Hugging Face's 15-trillion-token filtered Common Crawl derivative, notable because the team published ablations: they trained small models on each filtering variant to test whether a filter actually improved anything, rather than assuming it did. That methodology is the useful thing to take away, more than the corpus itself.

What filtered actually means

When a card says the data was filtered and deduplicated, it usually means some subset of these steps:

What "filtered and deduplicated" covers

Common Crawl snapshots	A few billion pages a month since 2008: boilerplate, spam, duplicates and every language mixed together
Language identification	The classifier is least accurate exactly where the data is thinnest, so low-resource languages disappear here
Deduplication, exact and near	Duplicated text is memorised far more readily, so this measurably reduces verbatim regurgitation
Quality classification	A small model trained to recognise reference-page text: a taste judgement wearing a statistical hat
Safety and toxicity filtering	The audit of C4 found a naive bad-word filter removing reclaimed language and text about LGBT topics
Benchmark decontamination	Only some labs, only the benchmarks they thought of, usually by n-gram overlap, which misses paraphrases

None of these steps is neutral and none is disclosed in enough detail to be reproduced for most commercial releases. A crawler opt-out added today changes nothing above this line, because those snapshots are already published and already mirrored.

- **Language identification**, keeping documents a classifier thinks are in the target languages. This step is where low-resource languages quietly disappear, because the classifier is least accurate exactly where the data is thinnest.
- **Deduplication**, exact and near-duplicate, at document and sometimes paragraph level. This matters more than it sounds: duplicated text is memorised much more readily, and dedup measurably reduces verbatim regurgitation.
- **Quality classification**, often a small model trained to recognise text that looks like a reference page or a textbook. This is a taste judgement wearing a statistical hat.
- **Safety and toxicity filtering**, with the collateral damage described above.
- **Benchmark decontamination**, removing documents that overlap with known test sets. Only some labs do it, only for the benchmarks they thought of, and usually by n-gram overlap, which misses paraphrases.

None of these steps is neutral, and none is disclosed in enough detail to be reproduced for most commercial releases.

Opt-out, and its honest limits

Several mechanisms now exist for saying do not train on this.

- `robots.txt` blocking specific crawler agents, including Common Crawl's `CCBot` and various lab-specific crawlers.
- The emerging `ai.txt` and `llms.txt` conventions, which are conventions rather than enforcement.
- Spawning's Do Not Train registry and the opt-out fields on some dataset hosts.

Each of these has the same limitation, and it is a mechanical one rather than a legal one. **They are prospective only.** Blocking a crawler today does nothing about the snapshots taken in the last fifteen years, which are already published, already mirrored, and already inside models that are already trained. There is no mechanism to remove a page from a crawl that has been distributed, and no mechanism to remove what a trained model learned from it short of retraining.

That is worth saying plainly rather than implying otherwise, because a lot of writing on this subject leaves people believing that adding a line to a file protects work that was published in 2019.

What you can do in practice

If you need to know what is in a model, you have three moves and they get progressively weaker.

1. **Search the corpus**, where the corpus is published. `grep` over Dolma or FineWeb is slow but conclusive for exact strings. Hugging Face's dataset search covers many public sets.
2. **Probe the model.** Prompt it with the first half of a document you suspect and see whether it completes the second half verbatim. This detects memorisation, which is a subset of inclusion; a model can have trained on something without memorising it.
3. **Test on fresh material.** Write new items in the same style and compare performance. A large drop on fresh items is evidence of contamination,

without ever seeing the corpus.

For your own datasets, do the thing the labs mostly do not: write a dataset card. Where each part came from, what licence it carries, what you removed, and the date. It takes an hour and it is the only version of provenance anyone downstream will ever be able to check.

BEFORE YOU MOVE ON

A publisher adds a rule to robots.txt in 2026 blocking Common Crawl's bot, then asks whether their archive from 2015 onwards is now protected from being used in model training. What is the mechanically accurate answer?

A. Yes, because Common Crawl honours robots.txt retroactively and removes previously crawled pages from published snapshots.

B. Yes for text, no for images, because image crawlers use different opt-out registries.

C. Only future crawls are affected; existing snapshots stay published, and models already trained on them cannot unlearn the content without retraining.

D. No, because robots.txt has no legal force, so the rule changes nothing at all.

6 · 10 min

The copyright question nobody has settled

THE ONE THING TO KEEP

Copyright and AI splits into three independent questions — whether training infringes, whether an output infringes, and whether an output can be owned — and they are answered differently in every jurisdiction and mostly still unresolved.

Why this lesson refuses to give you an answer

This is not legal advice, and no honest lesson on this subject in 2026 can end with a rule you follow. The law differs by country, several of the important cases are unresolved or under appeal, and legislatures are actively rewriting the ground. What you can have is the *shape* of the disagreement, which is stable enough to be useful: knowing which question is being argued lets you read a headline correctly and ask a lawyer a precise question instead of a vague one.

There are three separate questions, and most confused arguments are two people answering different ones.

Question one: is training itself an infringement?

Training copies material — into a crawl, into a filtered corpus, into GPU memory. Copying is the act copyright regulates. The dispute is whether this particular copying is permitted.

United States. The argument runs through fair use, and specifically through whether training is transformative and what harm it does to the market for the original. Courts have begun to split the question: some rulings have treated the act of training on lawfully obtained works more favourably than the act of assembling a library from pirated sources, which puts the spotlight on *ac-*

quisition rather than training. That distinction is doing a lot of work and is still being tested.

European Union. The DSM Directive created text and data mining exceptions. Research organisations get a broad one. Everyone else gets one that applies only where the rightsholder has not reserved their rights in a machine-readable way — which is what turns `robots.txt` and similar signals into something with legal weight in the EU that they do not have elsewhere.

Japan. Article 30-4 of its copyright act permits use of works for machine analysis in fairly broad terms, with limits where the use unreasonably prejudices the copyright owner. It is generally read as the most permissive of the major regimes.

United Kingdom. A narrow non-commercial research exception, repeated consultations on a broader one, and no settled position at the time of writing.

India. Fair dealing under section 52 is an enumerated list of purposes rather than an open standard like US fair use, which makes the American arguments non-transferable. This matters if you are building here.

Question two: what about the output?

Separately from how the model was trained, a specific generation can reproduce protected expression. Research has repeatedly shown that models emit training text verbatim under some conditions, and that the rate rises with duplication in the corpus and with model size. Where an output is substantially similar to a protected work, that is an ordinary infringement question and the training argument does not help you.

This is the question you actually control. Deduplicated training data reduces it. So does not prompting a model to reproduce a specific work, and so does checking output against sources when the stakes are high.

Question three: can the output be protected?

Whether what the model produces is itself copyrightable is a third, independent question. The US Copyright Office has taken the position that human au-

thorship is required, registering works where a human contributed enough and refusing purely machine-generated material. Other jurisdictions take different views; the UK has an unusual provision for computer-generated works. If your business depends on owning what you generate, this is the question to ask about, and it has nothing to do with the first two.

What this means for you, practically

- **Your exposure comes from what you do, not from what the lab did.** Using a model is a different act from training it. Most of the litigation is aimed at trainers and at people who assembled corpora, not at downstream users.
- **Open weights come with no indemnity.** Several hosted API vendors offer contractual indemnification for copyright claims arising from output. A file you downloaded offers you nothing of the kind. That is a real and underrated difference between renting and self-hosting, and it belongs in the comparison alongside cost and latency.
- **Keep records.** Which model, which revision, which licence, which date, what you generated and when. If a question arrives in two years, the record is the difference between an afternoon and a month.
- **Treat generated code and generated text differently.** Code has a strong culture of licence compliance and tooling to match; a snippet that matches a licensed repository is a familiar category of problem with familiar remedies.

> *If someone tells you this question is settled, ask them which of the three questions they mean, and in which country. The confident answers on both sides are usually answering a different question from the one you asked.*

BEFORE YOU MOVE ON

A team self-hosting an open model asks which copyright risk their choice of self-hosting changed compared with calling a hosted API. What is the most accurate thing to tell them?

A. Self-hosting removes the risk, because the copying happened at the lab and they only run the finished weights.
B. Self-hosting increases the risk of training-stage infringement, because running the weights repeats the original copying.
C. Nothing changed legally, because copyright attaches to the model file itself and travels with it to whoever holds it.
D. The output-similarity risk is unchanged, but they have given up the contractual indemnity several hosted vendors offer, so they now carry that exposure themselves.

7 · 9 min

What regulators ask of an open model

THE ONE THING TO KEEP

The EU's open-source exemption for general-purpose models still requires a copyright policy and a public training-data summary, does not apply above the systemic-risk compute threshold, and publishing a fine-tune can make you a provider with duties of your own.

The rules arrived, and they are not aimed where people assume

Most writing about AI regulation talks about frontier labs. A large part of the actual text applies to anyone who *places a model on the market*, and fine-tuning a model and publishing it can put you in that category. It is worth twenty minutes to know where the lines are, because the compliance burden for an ordinary team is small if you know about it in advance and unpleasant if you discover it later.

Two warnings before the substance. Details here move; check the current text rather than trusting a lesson. And this is orientation, not legal advice.

The EU AI Act, in the part that touches you

The Act regulates general-purpose AI models as a category of their own, with obligations that began phasing in during 2025. For a provider of a general-purpose model, the core duties are:

- **Technical documentation** of the model, its training and testing, kept and made available to authorities and to downstream providers who build on it.
- **A copyright policy**, including respecting rights reservations expressed in a machine-readable way — which is the clause that gives `robots.txt` style opt-outs legal teeth inside the EU.
- **A sufficiently detailed summary of the training content**, published, following a template the AI Office provides.

There is a partial exemption for models released under a free and open-source licence, and three things about it are regularly misreported.

1. It **does not cover the copyright policy or the training-content summary**. Those two survive the exemption. So an open release still owes a public data summary.
2. It **does not apply to models with systemic risk**, defined with a compute threshold around ten to the power of twenty-five floating point operations of training compute. That is a frontier-scale number; ordinary fine-tuning is nowhere near it.
3. The exemption depends on the licence genuinely being free and open source, which the community licences discussed earlier may not be. A bespoke licence with use restrictions and a user cap is not obviously within it.

When a fine-tune makes you a provider

This is the part teams miss. If you take an open base model, fine-tune it, and put the result on the market under your own name, you can acquire provider

obligations for the modified model. Guidance has generally focused those obligations on the modification you made rather than on the entire model, and there are thresholds below which a small tune is not treated as creating a new general-purpose model. But the direction is clear: publishing weights is an act with duties attached, and the answer is documentation you were better off writing anyway.

Separately, if you *use* a model in a high-risk application — employment, credit, education, essential services — deployer obligations apply regardless of whether the model is open, and those are considerably heavier.

Elsewhere

United States. No general federal statute. Sector regulators apply existing law, and several states have passed transparency and disclosure requirements. Export controls target advanced chips rather than weights, though proposals to cover model weights have been discussed repeatedly and none has settled.

China. Generative services offered to the public require filing and security assessment, and content labelling rules apply to synthetic media. This is why some Chinese-origin open models behave conservatively on certain topics: the domestic service obligations shape the tuning even where the weights themselves travel freely.

Everywhere. Content-provenance and labelling rules for synthetic media are the fastest-moving area, and they attach to the *deployer* more often than to the model.

A proportionate response

For a small team shipping a product on open weights, the whole of this reduces to a short routine.

- Keep the model inventory you already started: model, revision, licence, date accepted, where it runs.
- If you publish weights, publish a model card with intended use, evaluation, limitations and a description of training data. That satisfies most of what is asked and is good practice regardless.

- If you fine-tune on data you did not create, record where the data came from and under what terms.
- Decide, once, in writing, whether any of your uses fall in a high-risk category. Usually the answer is no, and having written down why is worth more than the sentence suggests.

> *The compute threshold is the number to remember, because it settles most anxiety. If your training run cost less than a house, you are not the model this clause was written about.*

BEFORE YOU MOVE ON

A team fine-tunes an 8B open model on 40,000 of their own support tickets and publishes the adapter on a public hub under a permissive licence. Which statement about their EU obligations is closest to right?

A. None apply, because the free and open-source exemption covers every obligation for anyone releasing under a permissive licence.

B. The systemic-risk rules apply, because publishing weights publicly is what triggers that tier.

C. Full technical documentation and third-party audit are required because they are now a provider of a general-purpose model.

D. They are far below the systemic-risk compute threshold and the open-source exemption relieves much of the documentation burden, but a description of what they trained on and a copyright position are still expected.

8 · 10 min

Trusting a file you did not build

THE ONE THING TO KEEP

Safetensors removes only one of the execution paths into your machine; trust_remote_code runs repository Python by design, and behavioural back-doors in weights or adapters are invisible to every scanner.

The threat is ordinary, not exotic

You are about to download several gigabytes from a stranger and run it on a machine that has your credentials on it. That is a supply-chain decision, and it deserves the same seriousness as adding a dependency to `package.json` — more, because the artefact is opaque and nobody reviews it line by line.

Five attack surfaces, in rough order of how often they actually matter.

1. Custom modelling code, which is the one people miss

New architectures appear before the libraries support them, so repositories ship their own Python and the loader is asked to run it:

```
model = AutoModelForCausalLM.from_pretrained(
    "some-org/some-new-model",
    trust_remote_code=True,          # this executes code from
the repo
)
```

That flag does exactly what it says. It downloads `modeling_*.py` and `configuration_*.py` from the repository and imports them. Safetensors protects the *weights* file; it does nothing about a Python file sitting next to it. A team that carefully avoids pickle and then sets this flag has closed the front door and left the side gate open.

The remedy is unglamorous and works: open the file and read it. These files are typically three hundred to a thousand lines of tensor operations, and anything that opens a socket, reads an environment variable or touches the filesystem stands out immediately. Pin the revision so the file cannot change under you afterwards.

2. Pickle-format weights

`.bin`, `.pt`, `.ckpt` are usually Python pickle, and unpickling executes code by construction. This is a known and exploited vector, not a hypothetical. Prefer `.safetensors`; keep `weights_only=True` when you must use `torch.load`; if the flag makes loading fail, treat that as the signal it is.

3. Names that are almost right

`meta-llama` and `meta-llarna` look identical in most fonts at small sizes. Organisation names on public hubs are first-come, and lookalike organisations get created. Copy repository identifiers from the model card or the official announcement, never from a blog post, and check the organisation has the verification badge where the hub offers one.

4. Poisoned adapters and backdoored fine-tunes

A LoRA adapter is a few hundred megabytes of weights that modify behaviour. Research has demonstrated backdoors that stay dormant until a trigger string appears, at which point the model produces attacker-chosen output — including insecure code, when the model is being used to write code. The model passes every benchmark you run, because your benchmarks do not contain the trigger.

There is no reliable scanner for this. The defences are procedural: prefer adapters from the organisation that made the base model or from a source you can attribute to a real team, evaluate on your own set, and treat a model used in a code path that writes to production as a privileged component.

5. Data poisoning upstream

Work published in 2025 found that a small absolute number of poisoned documents — a few hundred, not a percentage of the corpus — was enough to plant a backdoor across a range of model sizes. This is not something a downstream user can defend against directly. It is an argument for behavioural evaluation rather than provenance faith, and for keeping models out of positions where a single bad output causes an irreversible action.

The routine that actually gets done

Six lines, once per model, and then it is habit.

```
# 1. pin, and fetch only what you need
hf download Qwen/Qwen3-8B --revision 9a3f21c   --include
"*.safetensors" "config.json" "tokenizer*" --local-dir ./m

# 2. record what you got, so a change is visible later
sha256sum ./m/*.safetensors > ./m/SHA256SUMS

# 3. scan anything that is not safetensors
pip install modelscan && modelscan -p ./m

# 4. read any *.py in the repo before enabling trust_remote_-
code
ls ./m/*.py
```

Then three habits that cost nothing:

- **Download in a container with no credentials mounted and no network beyond the hub.** The first load is the dangerous moment.
- **Mirror what you depend on to your own storage.** It makes you independent of the upstream repository being edited, renamed or removed, and the `SHA256SUMS` file turns any later change into something you notice rather than something you inherit.
- **Put models in the inventory with the same seriousness as libraries.** Name, revision, hash, licence, where it runs, who approved it.

> *The honest summary: hub-side scanning is a filter, not a proof. It catches known-bad pickle patterns. It cannot detect a backdoor in the weights, because that is a behaviour, not a signature.*

BEFORE YOU MOVE ON

A team's policy is to download only .safetensors files, and they load a new architecture with trust_remote_code=True after confirming there are no .bin files in the repository. What is the flaw?

A. Safetensors files can still contain pickle payloads in their JSON header, so the format gives no protection.

B. trust_remote_code imports Python files from the repository, so the repo executes code regardless of the weight format the team checked.

C. The risk is acceptable because Hugging Face scans every uploaded repository, including the Python files.

D. The flaw is only in the missing revision pin; with a pinned commit, trust_remote_code is safe.

---

9 · 8 min

Vetting a stranger's fine-tune

THE ONE THING TO KEEP

Vet a community fine-tune on its licence chain, its named datasets, its stated limitations and its performance on your own examples, because download counts and benchmark tables without a harness carry almost no information.

Why this is a separate skill

Public hubs hold hundreds of thousands of fine-tunes. Some are excellent and were made by two people in an evening. Some are a merge of six other merges with a benchmark claim nobody can reproduce. The base models from major

labs come with an implicit accountability that community uploads do not, so you need a cheap procedure that separates the two without a week of work.

The procedure below takes about ten minutes and rejects most candidates in the first three.

Read the front matter first

```
hf download <org>/<model> README.md --local-dir ./card
sed -n '1,40p' ./card/README.md
```

You are looking for four fields and their absence is meaningful.

- `base_model` — names the parent. If it is missing on something that is obviously a fine-tune, the uploader either did not know or did not say, and both are bad signs.
- `license` — must be present, and must be consistent with the base. An Apache 2.0 tag on a Llama derivative is a mistake or a misunderstanding, and it means the uploader has not thought about the licence chain at all.
- `datasets` — names what it was trained on. Follow the link. A dataset that does not exist, is private, or is described as a proprietary mixture leaves you with nothing to evaluate.
- `language` — a claim you can test in five minutes and often the fastest way to catch an overstated card.

The claims that are worth nothing without a harness

A card that says the model scores 82 on some benchmark and does not name the harness, the number of shots, the commit and whether the score is single-sample is not lying, but it is not evidence either. Reproducing it is usually impossible, because the configuration is the missing half of the number.

Two specific patterns to treat as red flags rather than mere gaps:

- **A benchmark table that beats models ten times its size.** Occasionally real, usually contamination from a training set that included

benchmark-shaped data, or an aggregated score reported against other people's single-sample numbers.

- **A chain of merges.** Cards that describe a model as a merge of two models that were themselves merges. This is a real technique with real results, but the licence chain, the data provenance and the evaluation are all now unrecoverable, and any claim about what the model will not do is unsupported.

Signals that mean less than people think

Download counts measure how many people tried it, which is driven by whichever quantised repository got linked from a popular post. It is popularity, and popularity in this ecosystem moves in days. **Likes** are the same. **The name** is free: a model called something-Ultra-Instruct-v3 has told you nothing.

Recency is worth a little, because tooling compatibility improves and the field genuinely moves, but a two-year-old model from a lab beats a two-week-old merge for anything you have to support.

Ten minutes on a stranger's model card

Means less than people think	Means more than people think
Download counts, which track one popular link	An uploader with two years of consistent cards
Likes	A limitations section in their own words
A name containing Ultra, Instruct or v3	Files that agree: config, tokenizer, chat template
A benchmark table with no harness or shot count	A named dataset whose link actually opens
A score that beats models ten times its size	It beats the base on twenty of your examples

The right-hand column costs the uploader effort, which is exactly why it carries information. The left-hand column is free to produce, and in this ecosystem popularity moves in days.

Signals that mean more than people think

- **The uploader's history.** An account with twenty models over two years, consistent cards, and issues answered in the community tab is a different proposition from an account created last month.
- **A card that states limitations.** Anyone willing to write down what their model is bad at has evaluated it. This is the strongest single signal on

the page.

- **Files that make sense together.** `config.json` consistent with the claimed base, a tokenizer that matches, a chat template present. A GGUF-only repository with no source weights and no card is a conversion by a third party, which is fine, but the provenance runs through them as well as through the original.

The one training-data question that is not about quality

A large number of popular community fine-tunes were trained on outputs generated from a closed commercial API. Doing this typically breaches that API's terms of service, and no licence tag on the resulting weights repairs that. It also has a technical consequence people notice and misdiagnose: the model acquires the *style* of the teacher — the headers, the hedging, the closing summary — without acquiring the capability, so it reads better than it performs. If you are choosing on a quick read of a few outputs, this is precisely the failure that fools you.

The ten-minute verdict

Ask four questions and stop at the first no you cannot live with.

1. Can I trace the licence to a base I am allowed to ship?
2. Does the card name a dataset and a harness, or at least admit that it does not?
3. Does it beat the base model on twenty of *my* examples, run at the same settings?
4. Is there a maintained alternative from the organisation that made the base?

Question four is the one that saves the most time. Very often the answer is that the official instruct model, at the same size, does the job, and the fine-tune was solving a problem you do not have.

BEFORE YOU MOVE ON

A community fine-tune of a 7B base reports benchmark scores above several 70B models, has 400,000 downloads, and its card names no dataset or harness. What is the most likely explanation?

A. Its training data or evaluation configuration overlaps the benchmarks, and the download count reflects a popular link rather than verified quality.
B. Small models genuinely outperform large ones once fine-tuned on the right task, so the numbers are plausible as reported.
C. 400,000 downloads is strong independent verification, since problems would have been reported by now.
D. The missing harness only affects reproducibility, not the validity of the score itself.

10 · 9 min

Publishing weights people can trust

THE ONE THING TO KEEP

A trustworthy release is the base model's score reported next to yours under an identical, named configuration, plus a working chat template, an honest limitations section and a licence chain you actually have the right to grant.

The other side of every lesson so far

At some point you will fine-tune something worth sharing. Everything you have learned about vetting other people's releases is now a specification for your own: publish the things whose absence made you distrust somebody else.

The whole job is about four hours, most of it writing.

The files

Ship the same shape everyone else ships, because tooling assumes it.

```
config.json                     # architecture, matching the
base
generation_config.json          # eos_token_id, default sam-
pling
tokenizer.json / tokenizer_config.json   # including chat_tem-
plate
model.safetensors               # or sharded, never pickle
README.md                       # the model card
LICENSE
```

If you trained a LoRA, publish the adapter *and* say clearly which base revision it applies to. An adapter without a pinned base is a puzzle for whoever downloads it. Publishing a merged full-weight copy as well is a kindness for people whose serving stack cannot load adapters, at the cost of the disk.

If you publish a GGUF conversion, say which quantisation and from which source revision, because conversions are where silent quality loss enters the ecosystem.

The model card, in the order people read it

One paragraph: what it is and what it is for. Base model, what you tuned it on, the task you built it for. Somebody deciding in thirty seconds should be able to leave here.

Intended use and out-of-scope use. Be specific about out-of-scope. It is the section that protects both of you, and it is the one most people skip. If it is a support-ticket classifier in English and Hindi, say that it has not been evaluated on other languages rather than staying quiet.

Training data. Where it came from, how much, what licence or terms it carried, what you removed, and the date. If it contains anything personal, say what and say how you handled it. If you generated part of it with another model, say which model, because that carries terms of its own.

Evaluation. The number, and everything needed to reproduce it: the harness with its commit, shot count, decoding settings, and whether it is single-sample. Report the base model's score under the identical configuration next to yours — that comparison is the only one that shows what your work did, and a card that omits it invites the reader to assume the worst.

Limitations, in your own words. What it gets wrong, which inputs break it, where it is worse than the base. This section is the strongest trust signal on the page for exactly the reason you found it persuasive in other people's cards.

Licence and provenance. Your licence, the base model's licence, the naming requirement if the base has one, and a line saying what a downstream user inherits.

The bits that go wrong

The licence you cannot grant. A fine-tune inherits the base's terms. Tagging a Llama derivative Apache 2.0 does not make it Apache 2.0; it makes your card wrong. Where the base requires a name prefix or an attribution line, comply at the point of publishing, not after someone tells you.

A chat template that is missing or subtly different. If you changed the format during training, the template in `tokenizer_config.json` must match what you trained on, or every user gets degraded output and blames the model. Test it by loading your own repository fresh in a clean directory and running one generation. Half of broken community releases fail exactly here.

A missing or wrong `eos_token_id`. The symptom is generation that never stops, and it will be reported to you as the model rambles.

Weights that do not match the config. Load your own upload from scratch before announcing it. It takes four minutes and catches the shard that failed to upload.

Safety, without theatre

Say what tuning you did or did not do to refusal behaviour. If you removed safety tuning, say so plainly rather than describing the model as uncensored

and leaving the reader to guess what changed. Give a contact route for reports. Decide in advance what you will do if you learn the model is being used for something you find intolerable — usually the honest answer is that you can update the card and stop distributing it, and you cannot recall what has been downloaded. Saying that in the card is more useful than an acceptable-use policy you have no way to enforce.

> *The test for a good release is simple: hand your own card to the vetting checklist from the previous lesson. If your model would fail your own ten-minute review, fix the card before you announce it.*

BEFORE YOU MOVE ON

You publish a fine-tune with a card reporting 78% on your task's held-out set, the harness commit, the decoding settings and the shot count. A reviewer says the evidence is still incomplete. What is missing?

A. The training loss curve, which shows whether the model converged.

B. The number of parameters trained, which determines how much the model actually changed.

C. The base model's score on the same set under the identical configuration, without which 78% cannot be attributed to the fine-tuning.

D. Scores on public benchmarks such as MMLU, so the model can be compared with other releases.

CASE STUDY · MODULE 1

The MIT badge that was not theirs

A four-person education company in Pune, three weeks from shipping a doubt-solving app to Maharashtra state board students.

Ketki and her three colleagues had built the thing properly. Their app took a photograph of a maths problem from a Class 10 textbook, read it, and walked a student through the solution in Marathi or English. They had chosen a reasoning model because a wrong early step ruins a proof, and the distilled reasoning model they picked was good enough that two teachers at a pilot school had stopped correcting it.

The repository they had built on was `DeepSeek-R1-Distill-Llama-8B`. The DeepSeek papers were public, the R1 release was MIT, and MIT is about as free as a licence gets. Ketki had written "MIT" in the one-line technology section of the deck she showed schools, and nobody had asked a second question about it.

The second question arrived from a procurement officer at a chain of eleven schools in Nashik, who wanted a signed statement of every third-party component and its licence before a purchase order could be raised. It was a form. Her lawyer filled it in and then stopped at the model row and asked where the MIT text was.

It was not there. The repository's front matter named a `base_model`, and the base was Meta's. DeepSeek-R1 is MIT; a distillation into Llama weights is not, because the weights being distilled into are Meta's and Meta's community licence travels with them. Ketki had been reading the licence of the model that produced the training traces rather than the licence of the model she was actually running.

What the real licence asked for

None of it was onerous in isolation. The Llama community terms wanted "Built with Llama" displayed prominently. They wanted a specific attribution line in a notice file. They required that any model she derived from it have a

name beginning with "Llama" — and she had fine-tuned an adapter on four thousand solved problems and called the result Ganak. They attached an acceptable use policy that Meta could update after she shipped, and they stopped being free above seven hundred million monthly active users, which was not her problem and would be the first thing any acquirer's diligence checked.

The app icon, the splash screen and two thousand printed flyers for the Nashik schools all said Ganak. The flyers had gone to press eleven days earlier.

The two ways out, both expensive

Comply. Rename the model — not the app, only the model, which the terms actually constrain — to something beginning with Llama, add the attribution, put the badge in the About screen, reprint nothing. Cost: about a day of work, a conversation with the procurement officer explaining that the licence in her deck had been wrong, and a permanent dependency on a document another company can edit. Her lawyer's note was blunt: you are agreeing now to terms that can change later, and you cannot prove which version you accepted unless you keep a copy with a date.

Swap. Move to Qwen3-8B, which is Apache 2.0 with no naming rule, no use policy and no user cap. Cost: the prompt was tuned against the distilled model's habit of thinking before answering and did not transfer. The adapter would have to be retrained against a different base. Their forty-example evaluation would have to be rerun. Ketki's estimate was nine working days, which put them past the start of the school term, which was the only window that mattered that year.

The third option, which one of the four argued for at length, was to ship and deal with it later. The licence was not being enforced against anybody they knew of. The procurement form could say MIT and nobody would check.

What made it a decision rather than a choice

The argument that ended it was not about risk of enforcement. It was that they were about to sign a statement of components for a school, and the statement would be false. Ketki said she was not willing to put her name on it, and that settled the immediate question without settling the larger one, which was what they should be running a year from now.

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 complied for the launch and swapped afterwards. The model was renamed to Llama-Ganak-8B inside the app, "Built with Llama" went into the About screen and the notice file, and the procurement form was corrected before it was signed — which cost one awkward phone call and no schools. The nine-day swap to an Apache 2.0 base was scheduled for the gap between terms and took eleven days. What survived as a permanent change was smaller and more useful than either: a two-line-per-model inventory in the repository listing model, revision, licence, date accepted and where it runs, plus a copy of every licence and use policy at the version they accepted. When Meta's terms were next edited, they could see what had changed rather than guessing.

Ketki read the licence of DeepSeek-R1 and concluded her model was MIT. Which specific field would have caught the mistake in fifteen seconds, and why does that field exist?

The `base_model` field in the model card's YAML front matter. It names the parent a fine-tune was built from, and a fine-tune inherits the base model's licence — as does a fine-tune of a fine-tune. Reading the front matter and following `base_model` upward until a repository has no such field is the whole check. Its absence

on an obvious derivative is itself a signal: the uploader either did not know or chose not to say.

Her colleague argued that nobody enforces these terms and the form could say MIT. Setting aside enforcement, what does that argument cost the company later?

Three things. A false component statement to a customer is a representation the company has made in writing, and correcting it later is worse than correcting it now. The user-cap and naming clauses are exactly what an acquirer's or investor's diligence checks first, so the problem surfaces at the moment with the least room to fix it. And a team that does not know which licence it is under cannot answer what it may do with the model's outputs, which is the clause that decides whether they can ever generate synthetic training data from it.

The swap to an Apache 2.0 base was estimated at nine days, mostly prompt and adapter work. What does that estimate tell you about a decision they made much earlier?

That they had designed around a single family. Prompts do not transfer between families, and an adapter is bound to the base revision it was trained against, so the cost of moving was real — but nine days for an 8B swap is the lock-in that open weights were supposed to prevent. Keeping the prompt versioned per model, the evaluation harness model-agnostic behind one OpenAI-compatible endpoint, and a copy of the weights they depend on turns a nine-day migration into an afternoon plus a measurement.

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 DeepSeek-R1 is released under MIT. A team plans to ship DeepSeek-R1-Distill-Llama-70B commercially and files it in their inventory as MIT. What have they got wrong?

A. The base weights are Meta's, so the distill carries Meta's community licence
B. MIT permits commercial use only under a separate attribution agreement
C. A derivative carries no licence until its uploader selects one
D. Training on another model's reasoning traces transfers that model's MIT licence to the result

2 A customer's security review asks whether the benchmark a team quotes was in the model's training data. Which axis of the six-axis openness rubric do they need?

A. Licence, because a permissive licence grants the right to audit the corpus
B. Evaluation artefacts, because raw per-task outputs expose memorised answers
C. Checkpoints, because intermediate checkpoints reveal exactly when a benchmark score jumped
D. Training data, because only searching the corpus can answer it

3 A publisher adds CCBot to robots.txt in 2026 after finding their archive quoted verbatim by an open model. What does that achieve?

A. It removes the publisher's pages from existing Common Crawl snapshots
B. It makes the model unlearn the archive at its next fine-tuning run, because opt-out signals propagate through derivatives
C. Nothing for the trained model; it only affects crawls made after the change
D. It obliges the lab to retrain without those documents at the next release

4 A team refuses every pickle file and downloads only safetensors, then loads a new architecture with trust_remote_code set to true. What have they left open?

A. The tokenizer, because tokenizer.json is a pickle in several repositories
B. The flag imports Python from the repository, which then runs on their machine
C. Only the risk that a third party quantised the file without stating the source revision
D. Nothing; safetensors cannot execute code, so the whole load is now safe

5 Two community fine-tunes of the same base are candidates. One has two hundred thousand downloads and a benchmark table naming no harness. The other has nine hundred downloads, names its datasets and lists what it is bad at. Which is the stronger evidence?

A. The second, because writing down weaknesses means somebody actually evaluated it
B. The first, because a benchmark table is comparable across cards even without a harness
C. Neither, because only the base model's own evaluation transfers to a fine-tune
D. The first, because downloads aggregate many people's judgement of quality

6 A European team publishes a fine-tune under Apache 2.0 and assumes the AI Act's open-source exemption removes every general-purpose model duty. Which duties survive the exemption?

A. None, because a free and open-source licence removes every provider obligation

B. Only the systemic-risk duties, which attach to any published fine-tune regardless of scale

C. The technical documentation and a third-party conformity assessment

D. The copyright policy and the published training-content summary

MODULE 2

The map, and how to read it

Open weights are not only chat models. There are code models with their own token formats, embedding models that decide whether search works, rerankers, speech models, image models with the ecosystem's worst licence traps, regional models that beat much larger general ones in their own languages, and a market of providers serving all of it by the token. This block draws the map, names what is durable about each part, and tells you what a model of a given size can honestly be asked to do.

> BY THE END OF THIS MODULE YOU CAN
>
> Place any open release on a map of families, modalities, formats and serving options within ten minutes — naming its licence posture, its likely tokenizer, the tools that will serve it and the size class it competes in — and pick the right kind of model for a retrieval, code, speech or generation task rather than reaching for a chat model by reflex

---

11 · 9 min

The Families, and What Each Is Actually For

> THE ONE THING TO KEEP
>
> Families differ durably in licence, size ladder and ecosystem; whichever leads on quality changes every few months.

Any ranking of these families has a shelf life of about a quarter. What does not change that fast is the *shape* of each family: what sizes exist, what licence they

ship under, what tooling assumes them, and what they were optimised for. Choose on the durable properties, then measure.

Qwen (Alibaba) — the widest ladder

The most complete range of sizes anywhere: roughly 0.6B up to 235B, dense models and mixture-of-experts, with dedicated coder and math variants. The main Qwen3 line is Apache 2.0. Multilingual coverage is broad and Chinese-English is particularly strong.

Why it matters: there is a Qwen for the machine you actually have. You can prototype at 32B on a rented GPU and ship the 4B on a laptop with the same prompt format and the same tokenizer family. If you do not know where to start, start here.

Llama (Meta) — the ecosystem

Rarely the top of any chart now, and still the model most tools assume. Quantizations exist for everything. Fine-tuning recipes, serving configs, LoRA adapters, papers, Stack Overflow answers, and half the inference stack were written against Llama first.

Custom community licence with the naming, attribution and user-cap clauses from the previous lesson. Pick Llama when integration surface and the sheer weight of prior art matter more than a benchmark position.

Mistral (France) — permissive and practical

Most of the line is Apache 2.0. Efficient dense models, a strong 12B and 24B class, good European language coverage, and reliable function calling. Historically the family that made small models feel usable.

One caution: not every Mistral release is Apache. Codestral shipped under a non-production licence. Check per model rather than assuming the family posture.

DeepSeek — research in public, at frontier scale

The V3 mixture-of-experts line and the R1 reasoning line, with unusually detailed papers describing how they were trained. R1 is MIT, which is remarkable for a model at that capability level.

The honest caveat: the flagship is around 671 billion parameters with roughly 37 billion active. You are renting it, not running it. What most people actually run are the R1 distills into Qwen and Llama bases — which are good, are weaker than the teacher, and carry the base model's licence.

Gemma (Google) — quality per byte, with strings

Gemma 3 covers 1B, 4B, 12B and 27B, with vision on the 4B and up and long context on the larger sizes. Pound for pound this family tends to be excellent at the small end, which is exactly where most people are constrained.

The strings are the licence: custom terms, a prohibited use policy that Google can update, and an obligation to pass the restrictions downstream.

Phi (Microsoft) — synthetic data, small and sharp

Built on heavily curated and heavily generated training data rather than raw web scale. Phi-4 is 14B under MIT, with smaller and reasoning-tuned siblings. It scores strikingly well on reasoning and maths benchmarks for its size.

The known trade-off is worth stating plainly: a model trained mostly on textbook-shaped data has less long-tail world knowledge and can be more sensitive to prompts that do not look like its training. It rewards a narrow, well-specified task and punishes open-ended chat more than its benchmark numbers suggest.

Worth knowing about

- **OLMo (AI2)** — data, code, checkpoints and weights all released. The family to use if you need to study or defend how a model was trained.
- **Granite (IBM)** — Apache 2.0, enterprise framing, decent tool use, sizes tuned for cheap serving.

- **SmolLM (Hugging Face)** — very small, fully open, honest about being small.
- **Kimi, GLM, MiniMax** — large mixture-of-experts models with strong agentic and coding claims; typically too big to self-host.
- **Nemotron (Nvidia)** — re-tuned derivatives of other bases, optimised for Nvidia serving.

How to use this map

None of these holds a durable lead. What is durable: licence posture, the size ladder, ecosystem depth, language coverage, and house style. Narrow on those. Then test the two or three survivors on your own examples, because the family reputation tells you nothing about your task.

BEFORE YOU MOVE ON

A team must ship commercial on-device summarization in Portuguese and Hindi, on 8GB laptops with no discrete GPU. Which reasoning is soundest?

A. Shortlist small permissively licensed models that document those languages, then test both languages on real inputs, because family reputation says nothing about your specific languages.

B. Use DeepSeek-R1, since it is MIT licensed and sits at the top of the reasoning charts.

C. Use Phi, because it wins most small-model benchmarks at its size.

D. It hardly matters: any 8B model fits in 8GB of RAM, so choose on ecosystem depth alone.

12 · 8 min

Hugging Face as Infrastructure

> THE ONE THING TO KEEP
>
> Never load a pickle you did not create; prefer safetensors, and pin the commit rather than the branch.

Hugging Face is not a website with download buttons. It is a git host with large-file storage, an API, and a set of conventions. Once you see it that way, most of it becomes predictable.

A repo is a git repo

Every model is a git repository with git-lfs for the big files. It has branches, tags, and commit hashes. `main` moves. People push new quantizations, fix a broken chat template, or silently change a config.

So pin the revision:

```
pip install -U "huggingface_hub[cli]"
hf auth login

hf download Qwen/Qwen3-8B \
  --revision 9a3f21c \
  --include "*.safetensors" "config.json" "tokenizer*" \
  --local-dir ./models/qwen3-8b
```

The same in code: `from_pretrained("Qwen/Qwen3-8B", revision="9a3f21c")`. A pinned revision is the difference between a build you can rerun in six months and a build you cannot.

Three environment variables worth knowing, especially on a shared or small machine:

```
export HF_HOME=/data/hf                  # cache somewhere with
room
export HF_HUB_ENABLE_HF_TRANSFER=1       # faster parallel
downloads
export HF_ENDPOINT=https://hf-mirror.com # if the main host is
slow or blocked where you are
```

And clean up, because the cache silently grows to hundreds of gigabytes:

```
hf cache scan
```

The model card is a statement of intent

The README has YAML front matter — `license`, `base_model`, `language`, `pipeline_tag`, `tags` — and then prose. Read for six things: context length, the chat template, intended and out-of-scope use, what languages are claimed, what evaluations were run, and stated limitations.

A card that does not state a context length or show a chat template tells you something about how carefully the release was made. That is useful information even before you load anything.

The one hard rule about file formats

`.safetensors` is a JSON header describing tensor shapes and offsets, followed by raw bytes. There is no code in it. Loading it cannot execute anything. It also memory-maps, so it loads faster.

`.bin`, **`.pt`**, **`.pth`**, **`.ckpt`** are usually Python pickle. Unpickling *runs code by design* — that is what the format does. A crafted file opens a shell on the machine that loads it. This is not theoretical; malicious models have been found on public hubs. Hugging Face scans uploads, which is a filter, not a proof.

`.gguf` is llama.cpp's single-file format, carrying weights plus metadata plus the tokenizer. Data, not code. Parsers have had memory-safety bugs, so it is not magic, but it is in a different class from pickle.

Two classes of file in the same repository

Data, and safe to load

- safetensors: a JSON header, then raw bytes
- GGUF: weights, tokenizer and template in one file
- Memory-maps, so it starts before the file is read
- Nothing in the file can execute

Code, and it runs on load

- bin, pt, pth, ckpt: usually Python pickle
- Unpickling executes by design, not by bug
- modeling_*.py, run by trust_remote_code
- A failure at weights_only=True is the signal

Hub-side scanning is a filter, not a proof: it catches known-bad pickle patterns and cannot see a backdoor in the weights. If a repository has safetensors, use them. If it has only pickle, convert it yourself in a container with no credentials and no network.

In practice: if a repo has safetensors, use them. If it only has `.bin`, either find a converted mirror, convert it yourself inside a container with no network and no credentials, or skip the model. And keep PyTorch's safe default:

```
import torch
sd = torch.load("pytorch_model.bin", weights_only=True)  # re-
fuses arbitrary code; the default since torch 2.6
```

If loading fails with `weights_only=True`, that is a signal, not an inconvenience to switch off.

Sizes, so you can plan a download

Before you start a 140GB transfer on a metered connection, do the arithmetic. Roughly two bytes per parameter at bf16:

Model	bf16 files	Q8 GGUF	Q4_K_M GGUF
4B	~8 GB	~4.3 GB	~2.5 GB
8B	~16 GB	~8.5 GB	~4.9 GB
27B	~54 GB	~29 GB	~16 GB
70B	~140 GB	~75 GB	~42 GB

Fetch only what you need:

```
hf download bartowski/Qwen2.5-7B-Instruct-GGUF \
  --include "*Q4_K_M.gguf" --local-dir ./gguf
```

The rest of the hub

Datasets are the same git plus parquet arrangement. Spaces are hosted demos, which are useful for trying a model for thirty seconds before committing to a download. Inference providers route API calls to third parties serving those same open weights, which is how you test six models without downloading any of them — the subject of the last lesson.

BEFORE YOU MOVE ON

A colleague argues: "We pinned the exact commit hash and the repo passed Hugging Face's malware scan, so loading its pytorch_model.bin is fine." What is the flaw?

A. Pinning fixes which bytes you get, but a pickle still executes code when loaded, and a scanner is a filter rather than a guarantee.

B. There is no flaw: a pinned commit means the bytes cannot change, so the load is safe.

C. The flaw is performance: .bin files load more slowly and use more memory than safetensors.

D. The flaw is precision: .bin checkpoints are stored in fp32, so the model will use twice the memory it should.

13 · 9 min

Regional and specialist models

THE ONE THING TO KEEP

A language-focused model wins by knowing more of that language and spending fewer tokens on it, so compare tokenizer output as well as quality — and check the licence, because several of the best multilingual releases are non-commercial.

The map has more on it than six labs

The families in the previous lesson are the ones written about. They are not the whole ecosystem, and for a large share of the world they are not the right starting point either. Two other groups matter: models built for particular languages, and models built for particular domains.

The reason to care is mechanical rather than patriotic. A general model trained overwhelmingly on English carries two disadvantages in another language, and both are measurable. It has seen less of the language, so it knows less. And its tokenizer splits that language into more tokens, so every request costs more and every context window holds less. A model built for the language fixes both at once, which is why a 7B Indic-tuned model can beat a general 70B on Hindi summarisation while losing badly on English reasoning.

Language-focused families worth knowing

India. AI4Bharat, at IIT Madras, has released the most useful public work: IndicTrans2 for translation across 22 scheduled languages, IndicBERT, and speech models covering languages that commercial systems ignore. Sarvam AI builds Indic-tuned models on open bases and releases weights. Krutrim has released Indic models. The practical point for anyone working here is that translation, transliteration and speech in Indian languages are better served by these than by a general chat model, and the gap is largest in exactly the languages with the least web text.

South East Asia. SEA-LION from Singapore's AI Singapore covers Bahasa Indonesia, Thai, Vietnamese, Tamil and others with a tokenizer designed for them.

Arabic. Jais, from G42 and MBZUAI, was the first serious open Arabic-centric family. Falcon, from TII in Abu Dhabi, is Apache 2.0 and general rather than Arabic-only.

Europe. Teuken and the OpenEuroLLM effort target the 24 official EU languages, with the explicit aim of not being an English model with translation bolted on.

Multilingual generalists. Cohere For AI's Aya covers a very large number of languages and is genuinely good. Note the licence carefully: Aya models have shipped under CC-BY-NC, which is non-commercial. This is the single most common licence surprise in this part of the map, because the model is excellent and the announcement reads like any other open release.

Domain-specialised models

Medical. Meditron, OpenBioLLM, and several biomedical tunes exist and score well on medical question benchmarks. Treat every one of them as a research artefact. A model that answers medical exam questions well is being measured on a format, not on patient safety, and none of these releases has regulatory clearance for clinical use anywhere. The honest use is literature triage and drafting for a qualified reader.

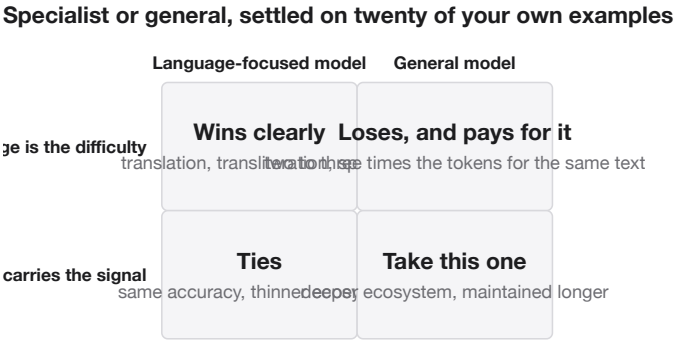
Legal. SaulLM and similar legal tunes. The same caution applies with an extra edge: legal answers are jurisdiction-specific, and a model tuned largely on one country's corpus will answer confidently about another.

Science and code. Protein language models, chemistry models, and the code families that get their own lesson next.

Enterprise-shaped. IBM's Granite is Apache 2.0, deliberately unexciting, and tuned for tool use and cheap serving. Nvidia's Nemotron models are retuned derivatives of other bases, optimised for Nvidia serving stacks.

How to decide whether a specialist is worth it

Run the general model and the specialist on twenty of your own examples. Three outcomes, and each has a clear response.



The bottom-left case is the common one and it argues for routing rather than choosing: the specialist behind the tasks that need it, the general model behind everything else. Two models on disk is not a problem; two models nobody measured is.

- **The specialist wins clearly.** Usually happens on translation, transliteration, speech, and anything where the language itself is the difficulty. Take the win.
- **They tie.** Usually happens on classification and extraction, where the task carries most of the signal and the language carries little. Take the general model, because its ecosystem is deeper and it will be maintained longer.
- **The specialist wins on the target language and loses everywhere else.** Common. This is an argument for routing: the specialist behind the tasks that need it, the general model behind everything else. Two models on disk is not a problem; two models nobody measured is.

Check the tokenizer as part of this. Encode a paragraph of your language in both and compare token counts. A specialist that produces 40% fewer tokens for the same text is 40% cheaper per request and holds 40% more context, before any quality argument.

> *The failure to avoid is picking a specialist because it is from your region rather than because it measured better. The reason these models often win is real and mechanical; the way to know it applied to your task is to run it.*

BEFORE YOU MOVE ON

A team in Chennai finds a 7B Indic-tuned model beats a general 70B on Tamil summarisation but loses on English reasoning. What does this most likely reflect?

A. The 70B was quantised more aggressively, so the comparison is unfair.

B. Summarisation is an easier task than reasoning, so smaller models close the gap on it generally.

C. The Indic model has seen far more Tamil and tokenises it more efficiently, while the general model retains its advantage where English data dominates.

D. Benchmarks for Indian languages are unreliable, so the Tamil result should be discounted.

14 · 9 min

Models that write code

THE ONE THING TO KEEP

Completion models trained with fill-in-the-middle and chat-tuned instruct models are different tools with different prompt formats, and code benchmarks like HumanEval are saturated enough that twenty diffs from your own repository are better evidence.

Two different jobs, two different models

Code models split into two kinds that people constantly confuse, and using the wrong one is the most common reason a local coding assistant feels broken.

Completion models finish the code around the cursor. They are trained with fill-in-the-middle, which means the file is cut into prefix, suffix and middle, and the model learns to produce the middle given both sides. The prompt is not a conversation; it is special tokens:

```
<|fim_prefix|>def parse_date(s):
    <|fim_suffix|>
    return dt<|fim_middle|>
```

Instruct models answer questions about code and write whole functions from a description. They expect a chat template, and asking one to do inline completion produces an explanation, an apology, and three code fences when what you wanted was four tokens.

If your editor plugin feels wrong, check which of these you have loaded. Every serious code family ships both, and the base or `-Base` suffixed repository is usually the one with fill-in-the-middle support.

The families

- **Qwen Coder.** The widest size ladder, from tiny to very large, Apache 2.0 on the main line, with fill-in-the-middle and repository-level training. If you want one answer, this is it.
- **DeepSeek Coder.** Strong, with a large mixture-of-experts version. The papers explain the repository-level data construction, which is worth reading if you care why these models handle imports across files.
- **Codestral (Mistral).** Good, and shipped under a non-production licence. It is the standard example of a model that people deploy without reading the terms.
- **StarCoder2 (BigCode).** Trained on The Stack v2, a corpus built with a public opt-out mechanism for repository owners, which makes it the most defensible provenance story in code. It ships under an OpenRAIL licence with use restrictions rather than a plain open source licence.
- **Granite Code (IBM)** and **CodeGemma (Google)** fill out the smaller end.

Sizes, honestly

At 1B to 3B you get autocomplete: finishing a line, filling an obvious body, completing a repetitive block. That is a genuinely useful product and it runs

on a laptop at usable speed, which is the whole reason it exists.

At 7B to 14B you get short functions from a description, test generation, and reasonable explanation of unfamiliar code. Expect to review everything.

Above that, the open models are increasingly competitive at whole-task work — fixing a failing test, making a small change across files — but this is the part of the field that moves fastest, and the honest advice is to measure on your repository rather than trust a number.

The benchmarks are the problem

HumanEval and MBPP are 164 and about 500 short self-contained problems, and both are saturated and heavily contaminated. A model scoring 90 on HumanEval tells you almost nothing about whether it can change your code.

The evaluations worth attention are the ones with an execution harness against real repositories — SWE-bench and its verified subset, LiveCodeBench with its rotating problems, and BigCodeBench for library use. They are harder to fake because something has to actually run.

The evaluation that decides your choice is smaller: take twenty diffs from your own repository history, give the model the surrounding code and the commit message, and see what it produces. An afternoon of that beats every leaderboard, because your repository has conventions no benchmark contains.

Wiring it into an editor, free

The free path here is complete and good.

- **Continue** is an open source extension for VS Code and JetBrains that points at any OpenAI-compatible endpoint, including a local `llama-server` or Ollama.
- **Tabby** is a self-hosted completion server with its own editor plugins.
- **llama.vim** and **llama.vscode** talk directly to llama.cpp.
- **Aider** works from the terminal against a git repository and applies diffs.

A typical laptop setup is a 1.5B to 3B model for inline completion, where latency under about 300 milliseconds is what makes it feel alive, and a larger model called on demand for chat. Run them as two separate endpoints; trying to serve both roles from one model is where the experience falls apart.

The two failures to expect

Imports and versions. Models write against the library versions that dominated their training data. Confident code calling an API that changed two releases ago is the single most common wrong answer, and it looks completely plausible.

Licence contamination in output. Generated code can reproduce distinctive chunks of training data. For anything you ship, the mitigation is the same as for human-written code: review, and for distinctive-looking blocks, search for them.

BEFORE YOU MOVE ON

A developer loads a 7B instruct code model into an editor plugin for inline autocomplete and finds that every suggestion arrives as a paragraph of explanation followed by a fenced block. What is the underlying cause?

A. The model is too large for low-latency completion, so the plugin is timing out and falling back to chat mode.

B. The plugin is sending a chat template when it should send a raw prompt, which any model would answer conversationally.

C. Instruct tuning trained the model to respond to a conversational turn, whereas inline completion needs a base model with fill-in-the-middle tokens that emits only the missing span.

D. The temperature is too high, so the model is elaborating instead of committing to a completion.

15 · 10 min

Embeddings you can host yourself

THE ONE THING TO KEEP

An embedding model is small, cheap and CPU-friendly, but its query and passage prefixes, normalisation, truncation limit and pooling are set by the model card, and getting any of them wrong degrades retrieval silently rather than raising an error.

What the model gives you

An embedding model turns a piece of text into a fixed-length list of numbers — 384, 768 or 1024 of them, typically — arranged so that texts with similar meaning sit close together. Closeness is measured with cosine similarity, which is the dot product of two vectors after both have been scaled to length one.

That is the whole idea, and it is the machinery under semantic search, retrieval-augmented generation, deduplication, clustering, recommendation and topic discovery. It is also the cheapest useful thing in this entire course: an embedding model is 20 to 600 million parameters, runs happily on a CPU, and embedding a hundred thousand paragraphs is minutes rather than hours.

The families

- **all-MiniLM-L6-v2.** 22 million parameters, 384 dimensions, Apache 2.0, and still the right default for a first system. It embeds thousands of sentences a second on a laptop CPU.
- **BGE (BAAI)**, **E5 (Microsoft)** and **GTE (Alibaba)** are the three families that trade the top of the retrieval charts. Small, base and large sizes; multilingual variants; permissive licences.
- **nomic-embed-text.** Apache 2.0, 8192-token context, which matters when your documents do not chunk neatly.

- **jina-embeddings.** Long context, multilingual, strong on code.
- **Qwen3-Embedding.** A recent family in several sizes with strong multilingual results.

Bigger is not automatically better here. The jump from MiniLM to a base-sized model is usually worth it; the jump from base to large often buys two points of recall for four times the compute and twice the storage.

The four details that decide whether it works

Prefixes. This is the mistake that silently halves quality. Several families are trained asymmetrically: queries and documents get different prefixes. E5 wants `query:` and `passage:`. BGE wants an instruction prefix on the query only. If you embed both sides identically, nothing errors and retrieval quietly gets worse.

```
from sentence_transformers import SentenceTransformer
m = SentenceTransformer("intfloat/multilingual-e5-base")
q = m.encode(["query: how do I cancel my order"], normalize_em-
beddings=True)
d = m.encode(["passage: To cancel an order, open ..."],
normalize_embeddings=True)
```

Read the model card for the exact strings. They differ per family and getting them wrong is undetectable without a measurement.

Normalisation. Cosine similarity assumes unit-length vectors. Most libraries offer `normalize_embeddings=True`; many vector databases assume you have already done it. Do it once, at write time, and be consistent.

Sequence length. Most embedding models truncate at 512 tokens, and truncation is silent. A 900-token chunk becomes an embedding of its first half, and the second half is unsearchable. Either chunk to fit the model or pick a long-context model deliberately.

Pooling. Whether the vector comes from the CLS token or the mean of all token vectors is decided by the model, not by you. Use the library's own wrapper rather than calling the transformer directly, or you will pick the wrong one.

Matryoshka, and cheap storage

Newer models are trained so that the first N dimensions of the vector are usable on their own. Truncating a 1024-dimension vector to 256 keeps most of the retrieval quality and cuts index size and search cost by four. If your card mentions Matryoshka representation learning, this is available; test the truncation on your own data rather than assuming the published loss figure transfers.

The arithmetic matters at scale. A million chunks at 1024 dimensions in float32 is 4 GB of index. The same million at 256 dimensions is 1 GB, and quantised to 8-bit integers it is 256 MB, which fits in memory on an ordinary server.

MTEB, and how to read it

MTEB is the standard embedding leaderboard, covering retrieval, classification, clustering and reranking across many datasets. Two cautions. Models are routinely trained on the training splits of MTEB tasks, so the leaderboard increasingly measures familiarity with those datasets. And the tasks are mostly English or translated English; performance in your language is a separate question the aggregate score hides.

The evaluation that decides your choice takes an hour: fifty real queries from your own logs, the correct document for each marked by hand, and recall at 10 measured for three candidate models. Whichever wins on your fifty queries wins, whatever the leaderboard says.

Where it goes wrong

Embeddings capture topical similarity, not logical relation. The sentence *the payment succeeded* and *the payment failed* sit very close together because they are about the same thing. Any system where the difference between two near-identical sentences matters needs something after retrieval to tell them apart — which is the next lesson.

BEFORE YOU MOVE ON

A team switches to an E5 embedding model, embeds queries and documents with identical code and no prefixes, and finds retrieval quality is worse than the MiniLM model it replaced. What is the most likely explanation?

A. E5 is asymmetric and expects distinct query and passage prefixes; without them the two sides are embedded into mismatched regions of the space.
B. E5 produces larger vectors, so cosine similarity becomes less discriminative at higher dimensions.
C. The MiniLM index was built with normalised vectors and the new one was not, so the scores are on different scales.
D. E5 is trained mainly on English, so it underperforms on any corpus containing other languages.

16 · 9 min

Rerankers, and why retrieval is two stages

THE ONE THING TO KEEP

A bi-encoder must compress each document before seeing the query, which is why retrieval is fast and imprecise; a cross-encoder reranker reads query and document together, which is why it is accurate and can only reorder what retrieval already found.

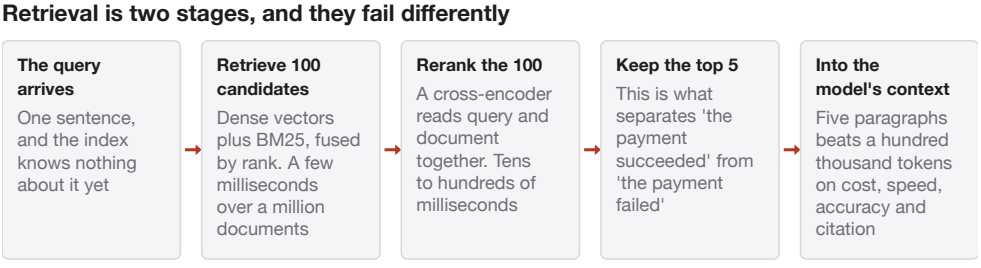
The limitation that creates the second stage

An embedding model has to compress a document into a fixed vector *before it knows what anybody will ask*. That is what makes search fast: you embed a million documents once, and each query is a nearest-neighbour lookup. It is also what limits quality, because the compression happens without the query.

A reranker removes that limitation by paying for it. It takes the query and one document *together*, runs them through a transformer, and outputs a single

relevance score. Every token of the query can attend to every token of the document. It is far more accurate and it is far too slow to run over a million documents.

So the standard shape is two stages:



A reranker cannot find what the first stage missed; it only reorders what it is given. So measure recall at 100 for the retriever first, then precision in the top five after reranking. Teams reporting one end-to-end number spend weeks tuning the wrong stage.

1. **Retrieve** the top 50 to 100 candidates with the embedding model, in a few milliseconds.
2. **Rerank** those candidates with the cross-encoder, in tens to hundreds of milliseconds, and keep the top 5 to 10.

The vocabulary: the embedding model is a **bi-encoder** because it encodes each side separately; the reranker is a **cross-encoder** because it encodes them jointly.

What it is worth

On typical document collections, adding a reranker moves the quality of the top 5 results more than upgrading from a small embedding model to a large one, and it costs less to change. In a retrieval-augmented system, the top 5 is what the language model sees, so this is usually the highest-return single change available.

The mechanism explains where it helps most: cases where two candidates are topically identical but only one answers the question. The pair *the payment succeeded* and *the payment failed* from the previous lesson is exactly the case a bi-encoder cannot separate and a cross-encoder can, because the cross-en-

coder sees the word *failed* in the presence of the word the user actually asked about.

The open rerankers

- **bge-reranker-v2-m3.** Multilingual, permissive, the reliable default.
- **Qwen3-Reranker.** Several sizes, recent, strong.
- **mxbai-rerank** and **jina-reranker** are alternatives worth measuring against.
- **ColBERT** and its successors take a middle path called late interaction: they keep one vector per token rather than one per document, and compare token to token. More accurate than a bi-encoder, faster than a cross-encoder, and much larger indexes. Worth knowing exists; worth using when the index size is affordable.

```
from sentence_transformers import CrossEncoder
ce = CrossEncoder("BAAI/bge-reranker-v2-m3", max_length=512)
scores = ce.predict([(query, doc) for doc in candidates])
top = [d for _, d in sorted(zip(scores, candidates),
reverse=True)][:5]
```

A base-sized reranker scores roughly 50 pairs in a few hundred milliseconds on a CPU and far faster on a GPU. Budget for it: reranking 100 candidates costs more than retrieving them by a factor of ten or more.

The other half of the first stage

Dense retrieval has a specific blind spot: exact strings. Product codes, error numbers, surnames, an unusual acronym. The embedding compresses these into a general neighbourhood, and the exact match is not recoverable.

BM25 — classic keyword scoring, available in every search engine and in small libraries — has the opposite profile. It finds exact tokens and misses paraphrases.

Running both and combining them is called hybrid retrieval, and the usual combination method is reciprocal rank fusion, which merges two ranked lists

by rank rather than by score, so you do not have to make two incomparable scores comparable:

```
score(d) = sum over lists of  1 / (60 + rank_of_d_in_list)
```

Hybrid plus a reranker is the configuration most production systems converge on, and it is not fashion; it is two different failure modes being covered by two different mechanisms.

The limit worth stating

A reranker cannot find what the first stage missed. It only reorders what it is given. If the correct document is not in the top 100 retrieved, no reranker will produce it, and every symptom will look like a reranker problem.

So measure the two stages separately. Recall at 100 for the retriever tells you whether the answer is even in the candidate set; that is the number to fix first. Then measure precision in the top 5 after reranking. Teams that report a single end-to-end number spend weeks tuning the wrong stage.

BEFORE YOU MOVE ON

A support search system returns the wrong article about 30% of the time. Recall at 100 for the retriever is 96%, and precision in the top 5 after reranking is 70%. Where should the work go?

A. Into the retriever, since 96% still means one query in 25 fails outright.

B. Into a larger embedding model, which would improve both stages at once.

C. Into the reranker or the ranking signals it uses, because the right document is nearly always in the candidate set and is being ordered below others.

D. Into chunking, since the top-5 failures indicate documents are being split badly.

17 · 9 min

Speech, in both directions

THE ONE THING TO KEEP

Whisper invents text during silence unless voice activity detection gates it, so a production transcription pipeline is VAD plus a fast reimplementation, and text-to-speech is where the open licences get genuinely restrictive.

Recognition: Whisper and the things built on it

OpenAI's Whisper is MIT-licensed, which makes it one of the most permissively released useful models in existence. The large models are around 1.5 billion parameters, and the family runs down to `tiny` at 39 million. It was trained on around 680,000 hours of multilingual audio and it transcribes, translates into English, and detects language.

You will rarely run the original implementation, because faster reimplementations exist and are equivalent in output:

- **faster-whisper** uses CTranslate2 and is roughly four times quicker with lower memory. This is the default choice on a server.
- **whisper.cpp** is the C++ port with the same relationship to Whisper that llama.cpp has to language models: it runs on a laptop CPU, a phone, or a Raspberry Pi.
- **distil-whisper** is a distilled English model, about six times faster than large with a small accuracy cost.

```
pip install faster-whisper
python - <<'EOF'
from faster_whisper import WhisperModel
m = WhisperModel("large-v3", device="cpu", compute_type="int8")
segments, info = m.transcribe("call.wav", language="hi", vad_-
filter=True)
for s in segments:
    print(f"[{s.start:.1f}-{s.end:.1f}] {s.text}")
EOF
```

The failure everybody meets

Whisper hallucinates on silence. Given a long pause, background noise, or a segment with no speech, it will confidently produce a sentence — very often a piece of text that was common in its training subtitles, such as a thank-you line or a channel sign-off. Studies of medical and legal transcription have found this in a meaningful fraction of segments, and it is worse on recordings with long silences, which is exactly what a phone call or an interview contains.

The fix is voice activity detection: run a VAD to find speech regions and transcribe only those. `vad_filter=True` above does this. It is not optional for production work, and it is the difference between a transcript you can trust and one that invents sentences.

Two smaller cautions. Word-level timestamps are approximate, derived from attention alignment rather than measured; treat them as within a couple of hundred milliseconds, not exact. And accuracy varies enormously by language: word error rates in the well-resourced European languages are in the single digits, while many Indian and African languages sit far higher. For Indian languages specifically, AI4Bharat's IndicWhisper and related work are usually better than the general model, for the same data-and-tokenizer reason as in the language-model case.

Diarisation — who spoke when — is a separate model. `pyannote.audio` is the common open choice; it is free but its models are gated and carry their own terms, so read them before shipping.

Synthesis: text to speech

The open TTS landscape is more fragmented and the licences are worse. Four worth knowing:

- **Piper.** MIT, tiny, runs in real time on a Raspberry Pi, dozens of languages. It sounds like a good 2015 voice, and for accessibility, announcements, IVR and anything on a low-power device it is the correct answer.
- **Kokoro.** Around 82 million parameters, Apache 2.0, and much better sounding than its size suggests. The current default for good quality at low cost.
- **XTTS (Coqui).** Voice cloning from a few seconds of audio, multilingual, and released under a licence that is not commercially free. Widely used anyway, which is a compliance problem waiting for someone.
- **F5-TTS**, **Orpheus** and others in the newer generation, generally permissive, higher quality, heavier.

The quality ladder here costs compute in a way that recognition does not: a good neural voice is several times more expensive per second of audio than transcribing the same second.

The consent question, stated plainly

Voice cloning from a short sample is now easy, cheap and open. Cloning a specific person's voice without their explicit consent is the harmful use of this technology, it is the basis of a large and growing fraud category, and several jurisdictions have started legislating on it. If you build with cloning, get recorded consent from the person whose voice it is, keep it, and label synthetic audio. This is not a legal opinion; it is the minimum that makes the work defensible.

A working pipeline

Audio in, transcript out, on a laptop, free: VAD to find speech, faster-whisper at `int8` for transcription, pyannote for speaker labels if you need them, then a small language model for summarising. Every piece of that runs offline, which is often the actual requirement — recordings of conversations are

among the most sensitive data most organisations hold, and the reason to self-host is usually not cost.

BEFORE YOU MOVE ON

A transcript of a one-hour interview contains several plausible sentences at points where the recording is silent. What is happening and what fixes it?

A. The audio has clipping artefacts the model is misreading as speech; re-encoding at a higher bitrate removes them.

B. The temperature fallback is producing random output; setting temperature to zero eliminates it.

C. The model is transcribing beyond the audio length; trimming the file to its true duration fixes it.

D. The model always produces text for whatever segment it is given, so silent segments get its most likely training phrase; gating with voice activity detection stops those segments reaching it.

18 · 9 min

Image and video weights, and their licence traps

THE ONE THING TO KEEP

One image-model family often ships variants under completely different licences, and the widely used FLUX.1 [dev] is non-commercial while [schnell] is Apache 2.0, so check the variant rather than the family before anything reaches a customer.

The part of the ecosystem with the worst licences

Image generation has the largest open community and the most confusing terms. A single release can ship three variants with three completely different

permissions, and the names do not tell you which is which. This lesson is mostly about not shipping something you were not allowed to ship.

The families

Stable Diffusion. SD 1.5 and SDXL are the models the community built on for years; the ecosystem of LoRAs, ControlNets and workflows around them is enormous. They ship under CreativeML OpenRAIL licences, which permit commercial use but attach use restrictions. Later Stability releases moved to a community licence with a revenue threshold above which a paid licence is required.

FLUX (Black Forest Labs). This is the trap worth memorising, because it catches people every week:

- **FLUX.1 [schnell]** — Apache 2.0. Genuinely open, commercially usable, fast, four-step generation.
- **FLUX.1 [dev]** — non-commercial licence. It is the one everybody uses and posts about, and it is the one you may not use commercially without a separate agreement.
- **FLUX.1 [pro]** — not released at all; API only.

Three variants, one family name, three different answers to may I ship this. The output images from the non-commercial variant are also covered by the terms, so generating with it and selling the picture is not a workaround.

Qwen-Image and several newer releases ship under Apache 2.0 with genuinely permissive terms, which is why they have become the default for anyone who has been bitten once.

Video. Wan, Hunyuan Video, LTX-Video and Mochi are the open video models. Licences vary from Apache to bespoke. Video is heavy: a few seconds at moderate resolution takes minutes on a good GPU and is not a laptop activity, though smaller distilled variants are appearing.

The free stack

Everything here runs on free software.

- **ComfyUI** is a node graph editor and the tool the community standardises on. It runs every model above, exposes each step, and its workflows are shareable JSON files. Steep at first and then the only thing you want.
- **Automatic1111** and **Forge** are the older form-based interfaces, easier for a first hour.
- **InvokeAI** sits between the two and has the better canvas for editing.
- **Krita with the AI diffusion plugin** puts generation inside a real painting application, which for retouching and inpainting is a better fit than any of the above.

Memory is the constraint. Roughly: SD 1.5 runs in 4 GB of VRAM, SDXL wants 8, FLUX at reduced precision wants 12 to 16, and video wants 24 and up. Quantised and distilled variants shift these down at a quality cost, and CPU generation works but is measured in minutes per image rather than seconds.

If you have no GPU: Google Colab's free tier, Kaggle's weekly GPU allowance, and Hugging Face Spaces running someone else's public demo will all generate images at no cost. For a small number of images the sensible answer is often to use a Space rather than to set anything up.

LoRAs, and what they are here

The adapter idea appears in image models too, and it is where most community work happens. A style LoRA is a few tens of megabytes trained on twenty to a hundred images, and it teaches a look, a character or an object. The licence of a LoRA is separate from the licence of the base model, and a LoRA trained on a living artist's work to reproduce that artist's style is the practice at the centre of the strongest objections to this technology. That is a genuine dispute about consent and livelihoods, not a settled question with an obvious answer, and it is worth being clear with yourself about which side of it a given piece of work sits on.

Provenance and the honest limits

Two things belong in any production use.

Label synthetic media. C2PA content credentials embed signed provenance in the file. Metadata is strippable and this is not a security control, but it is the emerging norm and several jurisdictions are moving to require disclosure of synthetic content.

Do not generate people who did not consent. Photorealistic images of identifiable real people, and sexual imagery of anyone without consent, are the two uses that have caused the most concrete harm from this technology. Every open model can produce them and no licence has ever prevented it. The constraint has to be yours.

BEFORE YOU MOVE ON

A studio generates marketing images with FLUX.1 [dev] locally, arguing that since the weights were downloaded freely and the images are their own creation, commercial use is fine. What is wrong with the reasoning?

A. Nothing, provided they do not redistribute the weights themselves, since licences govern distribution rather than use.

B. The [dev] variant carries a non-commercial licence covering the model and its outputs, so running it locally does not create a commercial right.

C. Images generated by a model cannot be owned by anyone, so the studio has no rights to sell regardless of the licence.

D. The problem is only the training data provenance, which is unresolved for every image model.

19 · 9 min

What a model of each size can honestly do

THE ONE THING TO KEEP

Stored knowledge scales with parameters and multi-step reliability compounds, so a small model fails on facts and long chains rather than on single narrow tasks — and narrowing the task usually beats increasing the size.

The question everyone asks second

After what should I run comes will it be good enough, and most published answers are either benchmark tables or enthusiasm. Here is a plain account, built from what these sizes reliably do rather than what they can be made to do in a demonstration.

Two mechanisms explain most of the pattern. **Stored knowledge scales roughly with parameter count**, because facts have to live in the weights. **Instruction adherence scales with tuning quality**, which is why a well-tuned 4B can follow a format better than a badly tuned 13B. And **multi-step reliability compounds**: if each step is right 90% of the time, five steps are right 59% of the time, which is why small models look fine on single questions and fall apart in agent loops.

What each size honestly does

70B and above	Rare knowledge, long chains, complicated specifications, agent work with many steps. Mixture-of-experts models live here
12B to 32B	Long-form writing that holds its shape, multi-step reasoning that mostly survives, much better multilingual. No longer a laptop
3B to 9B	Where most useful local work happens: schema extraction, summarising a page, answers from provided context, one well-formed tool call
1B to 3B	Code autocomplete, three fields out of an invoice, grammar and tone rewriting, short translation with a language-tuned model
Under 1B, after fine-tuning	Classification into fixed labels, intent detection, entity extraction, spam filtering, reranking. Thousands of items a second on a CPU

Stored knowledge scales with parameter count and multi-step reliability compounds, so a small model fails on facts and long chains rather than on single narrow tasks. For most things a 4B is failing, narrowing the task beats buying a 32B.

Under 1 billion

Not a chat model, whatever the demo shows. What this size does well, after fine-tuning on your data: classification into a fixed set of labels, intent detection, named entity extraction, spam and toxicity filtering, and reranking. These are real production jobs and a 300M model does them at thousands of items a second on a CPU.

Do not expect it to hold a conversation, follow a multi-part instruction, or know facts.

1 to 3 billion

Code autocomplete. Short structured extraction — pull the date, the amount and the supplier from an invoice. Simple classification without fine-tuning. Translation of short text, with a language-tuned model. Grammar and tone rewriting. On-device assistants with a narrow scope.

The failure to expect: it loses track of instructions with more than two or three clauses, and it confabulates freely when asked anything factual.

3 to 9 billion

The size where most useful local work happens. Reliable extraction into a schema, summarising a page, tagging, rewriting, answering questions from provided context, straightforward tool calls with a well-formed template, and decent conversation within a defined scope.

The characteristic failures are specific enough to plan around. Constraints decay with length — a 4B told to write in British English and avoid a word will comply for two paragraphs and drift by the fourth. Citations get invented when the source is not in front of it. Arithmetic beyond two or three steps is unreliable, and the model will present a wrong result with the same confidence as a right one.

12 to 32 billion

Longer-form writing that holds its shape. Multi-step reasoning that mostly survives. Reasonable code in unfamiliar libraries. Materially better multilingual behaviour, and much better adherence to a long list of constraints. This is where an open model starts to feel like a general assistant rather than a component.

The cost is that you have left laptop territory. A 27B at 4-bit needs roughly 16 GB of weights plus context, which means a well-specified desktop, an Apple machine with a lot of unified memory, or a rented GPU.

70 billion and above

Where open weights compete with commercial APIs on hard tasks: long chains of reasoning, rare knowledge, following a complicated specification, and agent work with many steps. Also where the mixture-of-experts models sit, which compute at the speed of a much smaller model but must still be held in memory in full.

The move that beats going bigger

For most tasks that a 4B is failing, the winning intervention is not a 32B. In rough order of return per hour spent:

1. **Narrow the task.** Three prompts each doing one thing beat one prompt doing three. A model that fails at classify-and-extract-and-summarise usually succeeds at each separately.
2. **Give it the information.** Most confabulation is a knowledge problem, and retrieval fixes knowledge problems that no size increase fixes reliably.
3. **Constrain the output.** Schema-enforced decoding removes an entire class of failure at every size.
4. **Show it three examples.** Few-shot prompting helps small models far more than large ones, because they have weaker priors about what you want.
5. **Fine-tune on 500 examples.** A tuned 3B beats an untuned 30B on a narrow task, routinely, and costs less to serve by an order of magnitude.

> *The reason to know these bands is that they let you say no to the wrong question. Somebody will ask whether a 7B can run their whole support system. The answer is that it can run four pieces of it well and one piece badly, and finding out which is which is an afternoon's work.*

BEFORE YOU MOVE ON

A 4B model handles one-step support replies well but succeeds only about 55% of the time on a five-step workflow, even though each individual step tests at around 88%. What does this indicate?

A. The model is losing the earlier steps from its context window, so a longer context would fix it.

B. Per-step reliability compounds, so 0.88 to the fifth power is about 0.53; the workflow needs to be split with checks between steps rather than a larger model.

C. The workflow prompt is badly written, since a model competent at each step should complete the chain.

D. Small models are unsuitable for any multi-step task and the work needs a 70B.

20 · 8 min

Renting open weights by the token

THE ONE THING TO KEEP

Providers serving the same open model differ in quantisation, context cap and default sampling, so a benchmark run against one provider does not transfer to another unless you pin the provider and record its serving configuration.

A market you should use before you buy hardware

Open weights are served commercially by a large number of companies: Together, Fireworks, DeepInfra, Groq, Cerebras, Novita, Nebius, Hyperbolic, and aggregators such as OpenRouter and Hugging Face Inference Providers that route to them. All of them speak the OpenAI chat format, so one script tests all of them.

This is the fastest path to a decision. Comparing six models across three providers takes an hour and no downloads. Buying a GPU first and then discovering that a 7B does not do your task is the expensive order to do things in.

The thing nobody tells you: the same model is not the same model

Two providers serving `Llama-3.3-70B-Instruct` can give you noticeably different quality, and neither is lying about which model they run. The differences come from serving decisions:

- **Quantisation.** One serves bf16, another fp8, another int4. Cheaper prices usually mean more aggressive quantisation, and it is often not stated on the pricing page.
- **Context limit.** A provider may serve a 128k model with a 32k cap, and silently truncate.

- **Default sampling.** Different default temperature, top-p and repetition penalty, applied when you do not specify. Always specify.
- **A modified system prompt or safety layer** inserted ahead of yours.

OpenRouter exposes the quantisation and context length per provider and lets you pin to a specific one, which is the reason to prefer an aggregator that shows this over one that does not. If you benchmark a model on one provider and deploy against another, you have not benchmarked what you deployed.

Price and speed have different shapes

Pricing is per million tokens, quoted separately for input and output, and output is typically two to four times the input price. Across the market, an 8B class model sits in the low tens of cents per million tokens, a 70B in the low dollars, and the very large mixture-of-experts models above that. The spread between the cheapest and dearest provider for the identical model is routinely three to five times.

Speed splits into two numbers that trade against each other. **Time to first token** is what an interactive user feels; **tokens per second** is what a batch job cares about. Specialist hardware providers such as Groq and Cerebras deliver output speeds an order of magnitude above GPU providers on supported models, which changes what is possible for anything interactive, at the cost of a much shorter model menu.

For batch work, none of that matters and you should sort by price.

Privacy, which is usually the real question

Policies differ and they are the reason many teams self-host despite the cost. Ask three questions of any provider before sending real data:

1. Are prompts and completions retained, and for how long?
2. Are they used for training, and is opting out the default or a setting?
3. Where does the request physically execute, and does that satisfy your data residency obligations?

Aggregators complicate this because your request passes through them to a third party, and the effective policy is the strictest link in the chain, not the one on the aggregator's page. Where an aggregator offers a no-logging setting, understand that it constrains which providers can serve you.

Free tiers, honestly

Several providers offer free credit or a free tier on smaller models, and Hugging Face gives a monthly credit allowance on inference providers. These are enough to evaluate half a dozen models properly and to run a hobby project. They are not enough to serve a product, and rate limits on free tiers make latency measurements meaningless.

How to use the market

- **Evaluate on a provider, deploy where the constraints say.** If data may leave, staying rented is usually cheaper than owning until volume is high.
- **Pin the provider and the quantisation**, and record both next to the model revision in your inventory.
- **Re-run your evaluation set after any provider change.** A silent switch to a cheaper serving configuration looks exactly like the model getting worse for no reason, and this is a common and confusing production incident.
- **Keep the weights anyway** if the model matters to your product. The provider may drop it; your copy will not.

BEFORE YOU MOVE ON

A team's summarisation quality drops noticeably one week with no change to their code, prompts or model name. Their aggregator routes to whichever provider is cheapest. What is the most probable cause?

A. The model publisher pushed an updated version under the same name.

B. Their input documents drifted, and the model is unchanged.

C. Routing moved them to a provider serving the same model at a lower precision or a shorter context cap.

D. Aggregate demand raised latency, and slower generation produces lower-quality output.

CASE STUDY · MODULE 2

The best model for Tamil, and the licence on it

A knitwear exporter in Tiruppur with forty staff, trying to sort three hundred WhatsApp messages a day into orders, sample requests, complaints and payment queries.

Vaanam Knits ships cotton jersey to buyers in Germany and the United Kingdom, and takes almost every domestic instruction on WhatsApp. The messages arrive in three registers that its two-person software team had to treat as three different problems: Tamil in Tamil script from the dyeing and stitching units, Tamil written in Roman letters from the sales staff, and English from the buyers' agents.

Suresh, who had built the company's order system and taught himself the rest, wanted the messages tagged automatically into four buckets so that a complaint never sat unread behind sixty delivery updates. He did the right thing and built an evaluation set first: sixty real messages pulled from a month of chat exports, twenty in each register, each labelled by the sales manager.

Three candidates and one clear winner

He tested three. A general 8B instruct model under Apache 2.0, running on the office machine. A 32B rented by the token from an aggregator, which was the expensive option and the one he expected to win. And an Indic-tuned 7B built on an open base by a lab whose Tamil work he had seen praised.

The Indic model won, and not narrowly. On Tamil script it was eleven points ahead of the general 8B. On Roman-script Tamil — the register nobody benchmarks and the one half his traffic used — it was fifteen points ahead, because the general model kept reading Tanglish as misspelled English and tagging a complaint about shade variation as a sample request.

The tokenizer told the same story from a different direction. Suresh encoded the same thousand messages with each tokenizer and counted. The Indic model spent 2.6 times fewer tokens on the Tamil-script messages than the general 8B did. That is not a quality argument; it is a bill, a latency figure and

an effective context length, all at once, and it was the number that convinced the managing director that this was engineering rather than enthusiasm.

The line in the front matter

The model card's front matter said `license: cc-by-nc-4.0`.

Non-commercial. Vaanam Knits is a business, the tagging would run inside a commercial operation, and no reading of the phrase made that a research use. There was no ambiguity to exploit and no lawyer to ask for a better answer, because the answer was on the page.

What was actually on the table

Ask for a commercial licence. Suresh wrote to the lab. The realistic view was that a forty-person exporter in Tiruppur was not the correspondent a research lab replies to quickly, and he had no date by which he could promise an answer.

Ship the general 8B anyway. Eleven and fifteen points worse, 2.6 times the tokens, and — this was the part the sales manager cared about — the specific failure was that complaints in Tanglish got mis-tagged, which is the one bucket the whole project existed to protect.

Route. Two models: the Apache-licensed general model on the English traffic, where it was as good as anything, and a separate, permissively licensed Indic model on the Tamil and Tanglish traffic. Suresh had found one, from a different family and a different lab, under Apache 2.0. It was four points behind the non-commercial model on Tamil script and six behind on Tanglish — worse than the winner, much better than the general model, and legal.

Do nothing, and keep the sales manager reading messages. The status quo is always an option and it had a real cost, which was that two complaints in the previous quarter had reached the buyer before they reached Tiruppur.

The tempting reading of this case is that the licence ruined a good technical decision. The more useful reading is that the licence was a property of the

model as much as its Tamil accuracy was, and Suresh found it in the fourth minute of a ten-minute check rather than in the fourth month of production.

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 routed. The Apache-licensed Indic model took the Tamil and Tanglish traffic, the general 8B took English, and a two-line rule in front of both decided which by script and by a short romanised-Tamil word list. The combined system scored four points below the non-commercial model on the sixty-message set and eighteen points above the single-general-model design. The letter to the lab was sent anyway and answered five weeks later with a pointer to a commercial tier that cost more than the whole project. What Suresh added to the company's checklist afterwards was one line, placed above the accuracy tests: read the front matter before you run the model, because a model you may not ship is not a candidate, however well it scores.

Suresh measured tokenizer fertility as well as accuracy. What three separate decisions did that one measurement inform?

Cost, because the bill is tokens multiplied by price and the general model spent 2.6 times as many on the same Tamil messages. Effective context length, because a window that holds a given number of tokens holds proportionally less of a Tamil conversation. And latency, because generation is per token, so a reply of the same length takes proportionally longer. None of these is a quality argument, which is why they persuaded a managing director who was not going to adjudicate an accuracy claim.

The routing design left them four points below the best model they had tested. Why is that often the right trade rather than a compromise?

Because the best model was not available to them at any price they could pay, so the real comparison was between routing and the single general model, where routing won by eighteen points. Routing also matches the structure of the traffic: the specialist is only better where the language itself is the difficulty, and on English the general model is as good and has the deeper ecosystem. The cost of routing is two models to keep measured rather than one, which is a real maintenance obligation and a much smaller one than a fifteen-point gap on complaints.

Half the traffic was Tamil written in Roman letters, and it was where the general model failed worst. Why is that register the one most likely to be missed in any model selection?

Because script, not just language, decides what a model saw in training, and romanised Indian languages live on web forums rather than in formal publishing, so coverage is uneven and unpredictable. No public benchmark covers code-switched romanised text, and a model card's language tag says nothing about script. The only way to know is to put real messages in the register your users actually write in into your own evaluation set — which is why Suresh's sixty messages were worth more than any leaderboard.

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 An Indic-tuned seven-billion model beats a general seventy-billion model on Hindi summarisation. Besides having seen more Hindi, which mechanism explains the win?

A. Smaller models are inherently better at summarisation than larger ones
B. Its attention layers were retrained for Devanagari, which uses a different positional scheme
C. Fine-tuning on Hindi increases the number of parameters allocated to Hindi
D. Its tokenizer splits Hindi into fewer tokens, so more of the document fits

2 A retrieval system on multilingual-e5-base returns worse results than expected. Nothing errors, and queries and passages were embedded with identical code. What is the likeliest cause?

A. Embedding models expect the query to be longer than the passage to score well
B. The index was built before every document had finished loading
C. E5 was trained asymmetrically and needs a different prefix on each side
D. The model needs more than 384 dimensions to separate these documents

3 A reranker has not improved answers. Recall at 100 for the first stage measures 0.52. What does that tell you?

A. The reranker is under-trained for this domain and should be replaced by a larger one

B. Almost half the questions have no correct document to reorder, so fix retrieval

C. The reranker should be run over the whole corpus rather than the top hundred

D. Cross-encoders cannot improve on a bi-encoder when the two share a base model

4 A transcript of a one-hour interview contains a thank-you line in three places where nobody spoke. What removes it?

A. Gate transcription with voice activity detection so silence is never decoded

B. Move from the base model to the large one, since larger models err less

C. Lower the temperature, because the insertions come from sampling rather than the audio

D. Split the audio into fixed thirty-second chunks so no segment is long enough to drift

5 A studio generates client artwork with FLUX.1 dev and argues that selling the images is fine because the images are their own creation. What is wrong with that?

A. FLUX is Apache 2.0, so the studio must publish its workflow files with the images

B. The studio needed the schnell variant, which is the only one released with weights at all

C. Nothing; a licence binds weights, and generated images are always the user's own

D. The dev variant is non-commercial and its terms cover the outputs too

6 A team benchmarks a seventy-billion model on one provider, then deploys against an aggregator's cheapest route. Quality drops with no code change. What most likely happened?

A. Benchmarks always regress in production because real inputs are longer than test inputs

B. The model was silently replaced with a smaller one from the same family

C. The cheaper route serves a harder quantisation with different sampling defaults

D. Aggregators add network latency, and slower responses are lower quality

MODULE 3

Inside the files

A model repository is a small number of files that between them fix the memory the model needs, the cost of every request in your language, the exact conversation format it will accept and the point at which it stops talking. This block reads those files properly: the config, the tokenizer, the chat template, the attention layout, the context claim, and the formats you convert between — so that you can predict a model's behaviour before you load it, and diagnose it from the files when it misbehaves.

> BY THE END OF THIS MODULE YOU CAN
>
> Open a model repository you have never seen and predict, from config.json and the tokenizer alone, its parameter count, its memory footprint at a given context length, its per-token cost in your languages and the exact conversation format it requires — then verify each prediction with a command and diagnose a runaway or garbled generation from the files rather than by guessing

21 · 8 min

Base, Instruct, Reasoning

THE ONE THING TO KEEP

Reasoning models buy accuracy on multi-step problems with tokens and latency; most production tasks are not multi-step.

One family ships three or four variants of the same size, and the suffix on the repo name decides whether your afternoon works. `Qwen3-8B-Base`, `Qwen3-8B`, `Llama-3.1-8B-Instruct`, `DeepSeek-R1-Distill-Qwen-7B`. These are not versions. They are different tools.

Base: a document continuer

A base model predicts the next token over the pretraining distribution. It has no assistant persona, no chat template, and no discipline about stopping. Ask it a question and it may reply with three more questions, because that is what a page of questions looks like on the web.

You prompt it as a document:

```
Product: Kotoba Wireless Earbuds
Review (5 stars):
```

Use a base model when you are: fine-tuning from scratch on your own data, doing few-shot classification by comparing token logprobs, measuring perplexity, or generating text that must not carry assistant voice.

A base model is not "the uncensored one". It is the unfinished one. It will refuse nothing and also follow nothing.

Instruct: base plus manners, and a format contract

An instruct model is a base model further trained on instruction-following examples and then on human or synthetic preferences. It expects an exact conversation format: special tokens, role markers, an end-of-turn token.

Get the format wrong and quality falls off a cliff in a way that looks exactly like "this open model is bad". This is the single most common self-inflicted wound in local inference. Print the template once and look at it:

```
from transformers import AutoTokenizer
tok = AutoTokenizer.from_pretrained("Qwen/Qwen3-8B")
print(tok.apply_chat_template(
    [{"role": "system", "content": "You are terse."},
     {"role": "user", "content": "Summarize TCP in two
lines."}],
    tokenize=False, add_generation_prompt=True))
```

If you serve through llama.cpp, vLLM or Ollama and call `/v1/chat/completions`, the template is applied for you from the model's own metadata. That is a good reason to use the chat endpoint rather than raw completions, even locally.

Reasoning: paying tokens for correctness

A reasoning model has been trained, usually with reinforcement learning against checkable answers, to produce a long chain of thought before committing. You see it as a `<think>` block or a separate channel in the response.

What you buy: multi-step maths, competition-style problems, non-obvious debugging, planning where a wrong early step ruins everything.

What you pay:

- Three to twenty times the output tokens for the same visible answer.
- Seconds to minutes of latency, which rules it out of anything interactive at small scale.
- More KV cache, so more memory, which matters on the machines most people have.

- Worse behaviour on short structured tasks. It will deliberate about "extract the invoice date", and sometimes reason itself away from the correct answer.
- Loop risk. Set a hard token cap.

Some families ship hybrids that toggle thinking on and off. Distills like `R1-Distill-*` are instruct models trained to imitate reasoning traces: cheaper, weaker than the teacher, often still worth it.

One practical detail: unless the model card says otherwise, strip previous `<think>` blocks before sending conversation history back. Feeding old chains in as context tends to degrade the next turn.

Choosing, in one pass

- Extraction, classification, routing, short replies → smallest instruct model that passes, temperature 0, constrained output.
- Writing in a particular voice → instruct, and judge by reading, not by scoring.
- Competition maths, algorithmic problems, root-cause analysis → reasoning, with a token cap.
- Fine-tuning on ten thousand of your own examples → base if you want full control of the output format, instruct if you want to keep its existing manners and just shift the domain.

The mistake to avoid is treating these as a quality ladder with reasoning at the top. They are three different contracts about what the model does with your prompt.

BEFORE YOU MOVE ON

A pipeline extracting invoice dates and totals is switched from an 8B instruct model to a 32B reasoning model. It gets slower and slightly less reliable. Best explanation?

A. The task is single-step and format-bound, so long deliberation adds latency and extra opportunities to overthink or break the required output shape.
B. The 32B model must be quantized too aggressively; requantizing at higher precision will fix the reliability drop.
C. Reasoning models are strictly stronger, so the prompt is missing an explicit instruction to think step by step.
D. Temperature 0 prevents the model from exploring alternative reasoning paths, so it should be raised.

22 · 10 min

Every number in config.json

THE ONE THING TO KEEP

config.json contains everything needed to derive a model's parameter count, its weight memory and its KV cache per token, and most of the parameters live in the feed-forward matrices rather than in attention.

Twelve numbers that decide everything

Here is a real configuration, lightly trimmed, from a Llama-3.1-8B repository:

```
{
  "hidden_size": 4096,
  "intermediate_size": 14336,
  "num_hidden_layers": 32,
  "num_attention_heads": 32,
  "num_key_value_heads": 8,
  "vocab_size": 128256,
  "max_position_embeddings": 131072,
  "rope_theta": 500000.0,
  "rms_norm_eps": 1e-05,
  "tie_word_embeddings": false,
  "torch_dtype": "bfloat16",
  "architectures": ["LlamaForCausalLM"]
}
```

Everything you need to size, serve and budget for this model is in that block. Learning to read it takes twenty minutes and pays for itself the first time somebody claims a model will fit on a machine where it will not.

Deriving the parameter count

Work per layer, then multiply.

Attention. Four projection matrices. The query projection is `hidden_size × hidden_size` = 4096 × 4096 = 16.8M. The output projection is the same, 16.8M. The key and value projections are smaller, because `num_key_value_heads` is 8 rather than 32: each is `hidden_size × (num_key_value_heads × head_dim)` where `head_dim = hidden_size / num_attention_heads = 128`. So 4096 × 1024 = 4.2M each, 8.4M for the pair.

Feed-forward. Three matrices in a gated design: gate, up and down, each `hidden_size × intermediate_size` = 4096 × 14336 = 58.7M. Together 176.2M.

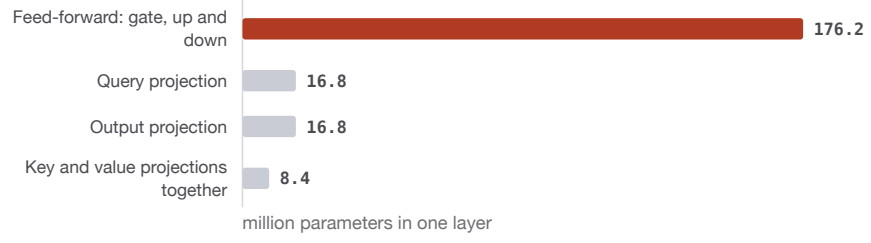
Per layer: 16.8 + 16.8 + 8.4 + 176.2 = 218.2M. Times 32 layers = **6.98 billion**.

Embeddings. The input table is `vocab_size × hidden_size` = 128256 × 4096 = 525M. Because `tie_word_embeddings` is false, the output head is a separate matrix of the same size, another 525M.

Total: 6.98 + 0.53 + 0.53 = **8.04 billion**, which is why it is called an 8B. At bfloat16, two bytes each, that is about 16 GB of weights.

Notice what the feed-forward layers did: 176 of the 218 million parameters per layer, roughly 80% of the model. Most of a transformer is the feed-forward stack, not the attention everybody talks about.

Where the parameters are, per layer, in Llama-3.1-8B



Two hundred and eighteen million per layer, of which the feed-forward stack is about eighty per cent. Most of a transformer is the part nobody talks about. The key and value projections are small because num_key_value_heads is 8 against 32 attention heads, and that same field makes the cache four times smaller.

What each number tells you operationally

`num_key_value_heads` **less than** `num_attention_heads` means grouped-query attention, which shrinks the KV cache by the ratio — here by a factor of four. This single field is the difference between a model you can serve to twenty users and one you cannot.

`vocab_size` matters far more at small sizes than at large ones. A 128k vocabulary costs 525M parameters at hidden size 4096. In a 1B model with hidden size 2048, a 256k vocabulary costs 524M of embeddings alone — half the model spent before any layer exists. That is the trade a large multilingual vocabulary makes, and it is why very small models often have small vocabularies and tokenise other languages badly.

`max_position_embeddings` is the maximum the architecture will accept, not a promise about quality. `rope_theta` at 500,000 rather than the older 10,000 is the signal that the positional encoding was set up for long context.

`intermediate_size` divided by `hidden_size` is usually between 2.7 and 4. An unusual ratio means an unusual architecture, and is worth a moment's attention before you assume standard tooling will load it.

`torch_dtype` tells you what the weights are stored as, and `architectures` tells your loader which class to build. An architecture name your library has never heard of is the situation that leads people to enable remote code execution, so it is worth noticing here rather than at the traceback.

Do it yourself in one command

```
python - <<'EOF'
import json, urllib.request
u = "https://huggingface.co/Qwen/Qwen3-8B/raw/main/config.json"
c = json.load(urllib.request.urlopen(u))
h, L = c["hidden_size"], c["num_hidden_layers"]
kv, heads = c["num_key_value_heads"], c["num_attention_heads"]
hd = h // heads
attn = 2*h*h + 2*h*(kv*hd)
mlp  = 3*h*c["intermediate_size"]
emb  = c["vocab_size"]*h * (1 if c.get("tie_word_embeddings")
else 2)
print(f"params ~ {(L*(attn+mlp)+emb)/1e9:.2f}B")
print(f"KV bytes/token @fp16 = {2*L*kv*hd*2:,}")
EOF
```

The second line is the number people most often get wrong and the subject of a later lesson. For this 8B model it is 131,072 bytes — 128 KB per token — so a 32,000-token conversation carries about 4.2 GB of cache alongside the 16 GB of weights.

> *The habit worth forming: before you download anything large, fetch the 2 KB config file and do this arithmetic. It costs nothing and it has stopped many pointless downloads.*

BEFORE YOU MOVE ON

Two 1B models are identical except that one has vocab_size 32000 and the other 256000, both with hidden_size 2048 and untied embeddings. What is the practical consequence?

A. The large-vocabulary model has far more transformer capacity, since a bigger vocabulary means more expressive layers.

B. The large-vocabulary model spends about a billion parameters on embedding tables, leaving much less for the layers, in exchange for tokenising more languages efficiently.

C. Vocabulary size affects only the tokenizer file, so the two models have the same parameter budget.

D. The larger vocabulary increases the KV cache proportionally, so long contexts cost four times as much.

23 · 9 min

Why the same sentence costs different amounts

THE ONE THING TO KEEP

Fertility — tokens per word — varies by a factor of three or more across languages and tokenizer vocabularies, and it multiplies straight into your bill, your effective context length and your latency.

Tokens are the unit of everything

Price is per token. Context length is in tokens. Speed is tokens per second. So the tokenizer — a lookup table and a merge algorithm, no neural network involved — quietly sets the cost of your product, and it does so differently for every language.

Almost every current model uses byte-level byte-pair encoding or a SentencePiece variant. The training procedure is simple: start from bytes, repeatedly merge the most frequent adjacent pair, stop at the target vocabulary size. Whatever was frequent in the tokenizer's training text gets a short representation. Whatever was rare gets chopped into pieces.

Vocabulary sizes across the families: Llama 2 used 32,000; Llama 3 moved to 128,256; Qwen uses around 151,000; Gemma uses 256,000. That progression is mostly about multilingual coverage.

The number to measure: fertility

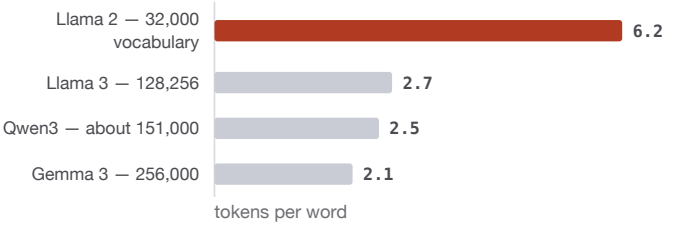
Fertility is tokens per word. Measure it on your own text, because it is the whole story:

```
from transformers import AutoTokenizer
text_en = "The delivery was delayed by three days and nobody
informed me."
text_hi = "डिलीवरी तीन दिन देर से हुई और किसी ने मुझे नहीं बताया।"

for name in ["meta-llama/Llama-2-7b-hf",
             "meta-llama/Llama-3.1-8B",
             "Qwen/Qwen3-8B",
             "google/gemma-3-4b-it"]:
    t = AutoTokenizer.from_pretrained(name)
    print(name, len(t.encode(text_en)), len(t.encode(text_hi)))
```

The pattern you will see: English costs roughly 1.2 to 1.4 tokens per word everywhere. Hindi in Devanagari on an older 32k tokenizer can cost five to eight tokens per word, because each Devanagari character is three bytes in UTF-8 and, with few merges learned for those byte sequences, the tokenizer falls back to something close to byte-per-token. A newer large-vocabulary tokenizer typically brings that down to two or three.

Tokens per word for the same Hindi sentence in Devanagari



English costs about 1.3 tokens per word on all four. At three times the fertility you pay three times per sentence on an identical price list, the same window holds a third of the document, and a reply of the same length takes three times as long. Measure this on your own text before you choose.

Three consequences follow directly, and none of them is about quality of writing:

- **Cost.** At three times the fertility you pay three times as much per sentence, on identical hardware and an identical price list.
- **Context.** A 32,000-token window holds perhaps 24,000 English words, or 6,000 Hindi words on a bad tokenizer. The same document does not fit.
- **Speed.** Generation is per token, so a reply of the same length takes three times as long.

This is the mechanical part of why a language-tuned model is often the right choice: the tokenizer usually comes with it.

Numbers, code and the details that surprise

Digits. Llama 3 splits numbers into groups of up to three digits; several other families split every digit separately. This is deliberate — consistent digit tokenisation makes arithmetic more learnable — and it means a long number is many tokens. It also means that arithmetic failures are partly a representation problem, not only a reasoning one.

Leading spaces. In byte-level BPE, `hello` and `hello` are different tokens. This is behind a whole family of odd bugs when you build prompts by string concatenation, and behind the trick called token healing that some libraries apply.

Code. Indentation matters: tokenizers trained with code merge runs of spaces, so four spaces can be one token or four depending on the family. On a

large codebase this is a real cost difference.

Whitespace and newlines are tokens too, which is why heavily formatted prompts with many blank lines cost more than they look.

The tokenizer is part of the model

You cannot mix them. Loading Qwen weights with a Llama tokenizer produces text that is grammatical nonsense, because token 5000 means something different in each. When you download a model, take `tokenizer.json` and `tokenizer_config.json` from the same revision as the weights.

This also constrains fine-tuning. You can add tokens to a vocabulary, but every added token needs a new row in the embedding matrix and in the output head, initialised somehow, and the model has never seen those rows. Adding vocabulary for a new language is a real technique and it requires continued pretraining to be worth anything — a later lesson covers it.

A five-minute exercise worth doing once

Take a thousand real documents from your own system. Run them through the tokenizers of the three models you are considering. Compute the mean tokens per document for each.

That number, multiplied by your request volume and the provider's price, is your bill. Teams routinely discover a 30% cost difference between two models of the same size that they were choosing between on quality alone.

BEFORE YOU MOVE ON

A team serving Hindi content moves from a model with a 32,000-token vocabulary to one with 151,000, keeping the same parameter count and the same per-token price. Costs fall by about 60%. Why?

A. The larger vocabulary lets the model answer more concisely, so it generates fewer tokens per reply.

B. Larger vocabularies allow more aggressive batching, which reduces the per-request cost on the provider's side.

C. The larger vocabulary learned merges for Devanagari byte sequences, so the same Hindi text becomes far fewer tokens.

D. The bigger tokenizer compresses the model's weights, so each token is cheaper to process.

24 · 9 min

The chat template is a contract

THE ONE THING TO KEEP

An instruct model was trained on an exact token sequence that ships with it in tokenizer_config.json, so build prompts with apply_chat_template and add_generation_prompt rather than by hand, and watch for a duplicated beginning-of-sequence token.

The most common self-inflicted wound

A team tries an open model, finds it rambles, ignores the system prompt and never stops, and concludes the model is poor. Nine times in ten the model is fine and the conversation was formatted wrongly. This lesson is how to check in thirty seconds.

An instruct model was trained on an exact byte sequence: special tokens marking where each role starts and ends, a specific ordering, a specific token

that means the assistant should now speak. Deviate and you are sampling from a distribution the model was never trained on.

Where the template lives

Inside `tokenizer_config.json`, in a field called `chat_template`, written in Jinja. It ships with the model, which means the model tells you its own format and you never have to guess:

```
from transformers import AutoTokenizer
tok = AutoTokenizer.from_pretrained("Qwen/Qwen3-8B")
print(tok.chat_template[:400])          # the source
print(tok.apply_chat_template(
    [{"role": "system", "content": "You are terse."},
     {"role": "user",   "content": "Summarise TCP in two
lines."}],
    tokenize=False, add_generation_prompt=True))
```

Print that once for every model you use. It takes ten seconds and it removes an entire category of mystery.

For a Llama 3 model the output looks like this, and every one of those markers matters:

```
<|begin_of_text|><|start_header_id|>system<|end_header_id|>

You are terse.<|eot_id|>
<|start_header_id|>user<|end_header_id|>

Summarise TCP in two lines.<|eot_id|>
<|start_header_id|>assistant<|end_header_id|>
```

The four things that go wrong

Forgetting `add_generation_prompt=True`. Without it the string ends after the user's turn and the model has not been told it is the assistant's turn. It will

often continue by writing another user message. This is the single most frequent cause of a model that talks to itself.

Double beginning-of-sequence tokens. The template already inserts `<|begin_of_text|>`. If you then call `tok(formatted_string)` with the default `add_special_tokens=True`, the tokenizer adds another. Two of them at the start measurably degrades output on some models and produces no error at all. Either use `apply_chat_template(tokenize=True)` and let it do both jobs, or pass `add_special_tokens=False` when tokenising a string the template already built.

Assuming a system role exists. Gemma's template has no system role. Passing one either raises an exception or is silently folded into the first user turn depending on the version. Several other models accept a system message but were tuned with almost none, and follow it weakly. Read the template: if `system` does not appear in the Jinja source, the model does not have one.

Hand-writing the format from a blog post. Formats change between versions of the same family. Llama 2 used `[INST]` markers; Llama 3 uses header tokens; they share nothing. The template in the repository is correct by construction; a remembered format is not.

Why the chat endpoint is the right default

When you serve through llama.cpp's server, vLLM or Ollama and call `/v1/chat/completions`, the server reads the template out of the model's own metadata and applies it. That is a strong reason to use the chat endpoint rather than raw completions even when you are running locally and it feels like an unnecessary layer.

The exception is when you deliberately want no template: few-shot classification against a base model, perplexity measurement, or fill-in-the-middle code completion. Then use the completions endpoint and construct the string yourself, knowingly.

Tool calling lives here too

Models that support function calling encode tools in the template, usually as a JSON block in a system turn or a dedicated tools section, with a specific token or format for the model's call. `apply_chat_template` accepts a `tools=` argument and will format them the way the model expects.

This is why tool calling that works on one model breaks on another with identical code: the format is per model, and a serving layer that does not implement that model's variant will produce calls the model was not trained to emit.

The thirty-second diagnostic

When output is bad, before changing anything else:

1. Print the exact string being sent, including special tokens.
2. Compare it to `apply_chat_template` output for the same messages.
3. Check for a duplicated start token at the beginning.
4. Check the string ends with the assistant header, not with the user's turn.

If all four are right, the problem is somewhere else and you have spent thirty seconds ruling out the likeliest cause.

BEFORE YOU MOVE ON

A locally served model keeps writing a plausible user question after answering, then answering that too. The chat template is applied correctly otherwise. What is the most likely omission?

A. The end-of-sequence token is missing from the tokenizer, so generation cannot terminate.

B. add_generation_prompt was not set, so the prompt ends after the user turn and the model continues the transcript rather than replying as the assistant.

C. The system prompt is too weak and needs an explicit instruction not to invent questions.

D. Temperature is too high, so the model wanders past the end of its answer.

25 · 10 min

Attention variants, and the cache that eats your memory

THE ONE THING TO KEEP

KV cache memory is two times layers times key-value heads times head dimension per token, it grows with every concurrent user, and grouped-query attention, cache quantisation and a lower context limit are the three levers that change it.

Why there is a cache at all

Generating token 500 requires attending to the keys and values of tokens 1 to 499. Recomputing them every step would be quadratic and absurd, so they are computed once and kept. That store is the KV cache, and on any serious deployment it is the thing that runs out, not the weights.

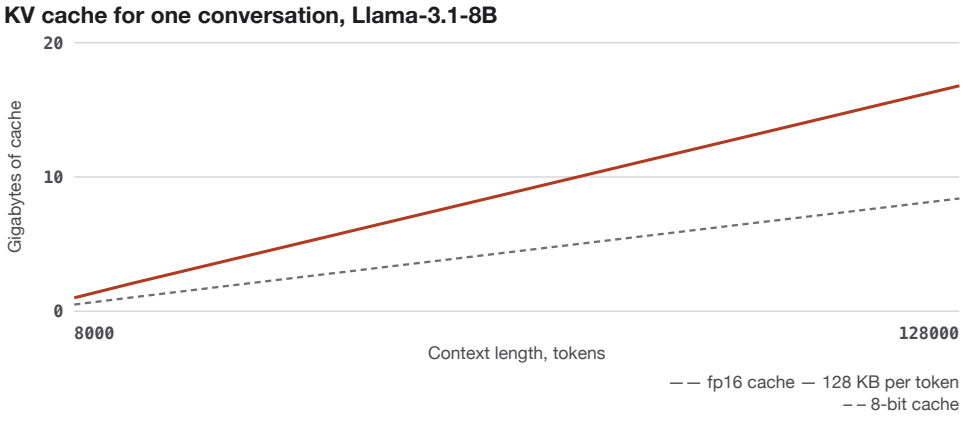
The size is exact and worth memorising:

```
bytes per token = 2 * num_layers * num_kv_heads * head_dim *
bytes_per_element
```

The leading 2 is for keys and values. For Llama-3.1-8B — 32 layers, 8 key-value heads, head dimension 128, fp16 — that is 2 × 32 × 8 × 128 × 2 = **131,072 bytes per token**, or 128 KB.

- 8,000 tokens of context: about 1.0 GB
- 32,000 tokens: about 4.2 GB
- 128,000 tokens: about 16.8 GB, which is as much as the weights

And that is for **one conversation**. Ten concurrent users at 8k context each need 10 GB of cache on top of the model. This is why a server that runs beautifully in a demo falls over with twelve people on it.



The weights are 16 GB. At 128k of context the cache alone matches them, and this is one conversation: ten concurrent users at 8k each want 10 GB on top of the model. Quantising the cache halves it, at a cost that shows up on long-context retrieval before it shows up on chat.

Three attention layouts

Multi-head attention gives every query head its own key and value heads. Maximum quality, maximum cache. Llama 2's 7B has 32 query heads and 32 key-value heads, so its cache is 512 KB per token — four times the Llama 3 model above, at a similar parameter count.

Multi-query attention gives all query heads a single shared key-value head. Smallest possible cache, some quality cost.

Grouped-query attention is the compromise everything now uses: several query heads share one key-value head. The ratio `num_attention_heads / num_key_value_heads` is the saving. At 32 and 8, the cache is a quarter of what full multi-head would need, with a quality difference small enough that the whole field adopted it within a year.

This is why `num_key_value_heads` is the field to look at in a config when you care about serving. Two models of identical parameter count can differ by four times in how many users one GPU will hold.

Sliding window attention attacks the problem differently: each token attends only to the last N tokens. Mistral's early models used a 4,096-token window; Gemma alternates local sliding-window layers with occasional global layers, so most layers keep a small fixed cache while a few see everything. The cache stops growing past the window, which turns a linear memory cost into a

constant one. The trade is that information from far back reaches the present only indirectly, through several layers of local attention.

What you can actually do about it

Quantise the cache. Most serving stacks can store keys and values in 8-bit rather than 16-bit, halving the memory for a small quality cost that is usually invisible on ordinary tasks:

```
llama-server -m model.gguf -c 32768 --cache-type-k q8_0 --
cache-type-v q8_0
vllm serve Qwen/Qwen3-8B --kv-cache-dtype fp8 --max-model-len
32768
```

Test it before you trust it. Cache quantisation hurts more on long-context retrieval than on short chat, because subtle key distinctions are what let the model find a specific fact far back.

Cap the context you actually allow. Serving a 128k model with `--max-model-len 8192` is not cowardice; it is choosing to serve sixteen times as many users. Most production traffic never needs more.

Let the serving layer manage it. vLLM's paged attention allocates the cache in fixed blocks like operating-system pages instead of reserving a contiguous maximum per request. That removes most of the waste from over-allocating for requests that turn out short, and is a large part of why vLLM serves several times more concurrent users than a naive implementation on the same hardware.

Reuse the shared prefix. If every request starts with the same 2,000-token system prompt, prefix caching computes it once and shares it across requests. On a chat product this is often the single biggest saving available, and it is a flag rather than a project.

The arithmetic to do before you buy

Take your GPU memory. Subtract the weights at your chosen precision. Subtract about 1 to 2 GB of runtime overhead. Divide what remains by (bytes

per token × your context limit). That is your concurrency.

For a 24 GB card running the 8B at Q4 with 8k context: 24 − 5 − 2 = 17 GB, divided by 1.0 GB per conversation, is about 16 concurrent conversations. If that number is smaller than your traffic, the fix is a shorter context limit or cache quantisation long before it is a bigger model.

BEFORE YOU MOVE ON

Two 7B models have the same layer count and head dimension. Model A has 32 key-value heads, model B has 8. On the same 24 GB GPU at the same context length, what differs?

A. Model A is four times faster to generate with, since more key-value heads mean more parallel attention work.
B. Nothing meaningful, since both models have the same parameter count and therefore the same memory footprint.
C. Model B serves roughly four times as many concurrent conversations, because its KV cache per token is a quarter the size.
D. Model A supports a four times longer context, because more key-value heads give it more positional capacity.

---

26 · 9 min

What a 128k context window really means

THE ONE THING TO KEEP

Advertised context comes from extended positional encodings, but the effective length on tasks harder than finding one sentence is often a fraction of it, and prefill cost grows with the square of prompt length.

Advertised, supported, effective

Three different numbers travel under the same name.

Advertised is what the announcement says: 128k, 256k, a million. It comes from `max_position_embeddings` and from whatever context extension the lab applied.

Supported is what your serving stack will actually run, which is set by your memory. The previous lesson's arithmetic decides this, and for most people it is between 8k and 32k regardless of what the model claims.

Effective is the length at which the model still does the task well. It is usually far below the advertised number, and it is the only one that matters.

Why the gap exists

Position information in current models comes from rotary embeddings, which rotate the query and key vectors by an angle proportional to position. The rotation frequencies are set by `rope_theta`. A model trained with `rope_theta` at 10,000 over 4k-token documents has never seen the rotations that correspond to position 90,000, and its behaviour there is extrapolation.

Labs extend context in one of three ways. **Linear interpolation** squeezes positions into the trained range, at some cost to fine positional resolution. **NTK-aware scaling** and **YaRN** adjust the frequencies unevenly so that high-frequency components, which carry local ordering, are disturbed least. All of them are usually followed by a short continued-pretraining run on long documents, and the amount of that training decides how real the extension is.

So a model can honestly advertise 128k and be trained on very few genuinely long documents. Nothing breaks at 100k; it just gets vaguer.

Needle tests pass, and prove less than they look

The standard demonstration hides a sentence in a long document and asks the model to find it. Modern models pass this at very long lengths, and the resulting green charts are what the announcements show.

Retrieving one distinctive sentence is close to the easiest possible long-context task. Harder evaluations — multiple needles, tracking a variable through a long trace, aggregating facts spread across the document, or answering when

the answer is absent — show much shorter effective lengths. Published work along these lines has repeatedly found models whose effective length on demanding tasks is a quarter or less of what they advertise.

There is also a well-documented positional effect: material at the start and end of a long context is used more reliably than material in the middle. Put the instruction and the most important document at the edges, not buried at 60%.

The costs nobody quotes

Prefill is quadratic in attention. Processing a 100,000-token prompt is not twelve times a 8,000-token prompt; the attention part grows with the square. Time to first token on a very long prompt can be tens of seconds on hardware where short prompts feel instant.

The cache is linear and large. 128k of context on the 8B model from the last lesson is about 17 GB of KV cache, which is more than the weights.

Rented long context is expensive. You pay per input token, every request. A 50,000-token prompt sent a thousand times a day is 50 million input tokens daily, and at typical prices that is a real monthly bill for something a retrieval step would have reduced to 3,000 tokens.

What to do instead, mostly

For nearly every application, retrieving the right 3,000 tokens beats sending 100,000. It is cheaper, faster, more accurate, and it gives you a citation trail. The exceptions are genuine: a single document that must be reasoned over as a whole, a long code file, a legal contract where any clause may matter. Those are real, and they are rarer than long-context marketing implies.

When you do need it, three habits help. Put instructions at the end as well as the beginning. Number or label the sections, so the model can refer to them. Ask for the location of the evidence along with the answer, which both improves accuracy and gives you something to check.

Measure your own effective length

Take five real documents at your longest realistic size. Ask three questions of each whose answers sit at different depths — near the start, in the middle, near the end. Score the answers. Then halve the document length and repeat.

If accuracy at half the length is much better, your effective length is below where you are operating, and no amount of prompt work will fix that. This takes an hour and is the only version of this number that describes your material.

BEFORE YOU MOVE ON

A model advertised at 128k passes a single-needle retrieval test at 120k but fails a task requiring facts from four separate places in a 60k document. What does this indicate?

A. The model's positional encoding is broken above 60k, so it should be run at half the advertised length.

B. The prompt is too long for the KV cache, so earlier tokens are being evicted.

C. Single-needle retrieval is a much easier task than aggregating dispersed facts, so the effective length for real work is well below the advertised figure.

D. The four facts are being lost to tokenisation differences across the document.

27 · 8 min

Mixture of experts, and the memory it still needs

THE ONE THING TO KEEP

A mixture-of-experts model computes like its active parameter count and occupies memory like its total, so it is outstanding on a machine with spare memory and useless on one without.

Two parameter counts, and only one of them is speed

A mixture-of-experts model replaces the single feed-forward block in each layer with many of them — the experts — plus a small router that picks a few per token. A model described as 30B-A3B has around 30 billion total parameters and around 3 billion active for any given token.

The consequence people want is real: **compute scales with active parameters**. A 30B-A3B model generates at roughly the speed of a 3B dense model, which is a large multiple faster than a 30B dense model would be.

The consequence people forget is equally real: **memory scales with total parameters**. The router can send the next token to any expert, so every expert must be resident. That 30B-A3B model needs about 18 GB at 4-bit quantisation, the same as a 30B dense model. You get 3B speed only if you can hold 30B.

That single sentence decides whether a mixture-of-experts model is the best or the worst choice you could make. On a 32 GB workstation it is excellent. On an 8 GB laptop it is unusable, while a 4B dense model works fine.

How the routing works

For each token, at each layer, the router scores the experts and selects the top few — commonly two of eight, or eight of a hundred and twenty-eight in the

larger designs. Only those run. Routing is learned during training, and it does not divide neatly by topic: experts do not correspond to physics and poetry, whatever the name suggests. They are learned specialisations of a kind nobody has cleanly interpreted.

Some architectures add a shared expert that runs for every token, holding the general capability while the routed experts specialise.

Training these models is harder than training dense ones, because the router can collapse onto a few popular experts and leave the rest untrained. Load-balancing losses exist to prevent that, and they are a large part of what the DeepSeek papers describe in detail.

What you meet in practice

The open MoE models include Qwen3's 30B-A3B and larger variants, DeepSeek's V3 line at around 671B total with roughly 37B active, and Mixtral's earlier 8x7B and 8x22B. The very large ones are rental propositions, not self-hosting ones; the DeepSeek flagship needs several hundred gigabytes of memory before any context.

Two operational quirks are worth knowing.

Batching is less efficient. In a batch, different tokens route to different experts, so the server ends up touching most of the weights anyway. Throughput gains at high concurrency are smaller than the single-stream speedup suggests.

Expert offloading works, slowly. llama.cpp and others can keep some experts in CPU memory or on disk and fetch them as needed. It makes a model that would not fit run at all, at a speed that depends entirely on the transfer rate. Useful for batch work overnight, painful interactively.

The choice, stated plainly

Ask one question: how much memory do I have?

- **Plenty of memory, want speed.** MoE is the best value in the ecosystem. A 30B-A3B on a machine with 32 GB gives you close to 30B-class

quality at close to 3B-class speed.

- **Memory constrained.** Dense, every time. A 4B dense model that fits beats a 30B-A3B that does not, and the comparison is not close, because the alternative to fitting is not running.
- **Renting.** It barely matters to you; the provider's price already reflects their side of this trade, and MoE models are often priced attractively for their quality precisely because they are cheap to serve.

> *The recurring mistake is reading 30B-A3B as if the model were 3B. Read it as a 30B model that runs quickly. The first number is the one your hardware has to satisfy.*

BEFORE YOU MOVE ON

Someone with a 16 GB laptop reads that a 30B-A3B model has only 3B active parameters and expects it to run like a 3B model. What will actually happen?

A. It will run at 3B speed but produce 30B-quality output, since only the active experts matter at inference.

B. It will run correctly but use four times the memory of a 3B model, since only the routed experts are loaded.

C. All experts must be resident because routing is per token, so around 18 GB at 4-bit will not fit and the model either fails or offloads to disk and crawls.

D. It will work, because mixture-of-experts models quantise better than dense models of the same total size.

28 · 9 min

Models that can see

THE ONE THING TO KEEP

A vision-language model turns image patches into tokens through a projector, so images consume context and cost proportional to resolution, and the reliable failures are counting, dense layout and small text rather than description.

The architecture, in one paragraph

A vision-language model is three pieces glued together. A **vision encoder**, usually a SigLIP or CLIP-style transformer, turns an image into a grid of patch embeddings. A **projector**, often just a two-layer network, maps those into the language model's embedding space. The **language model** then treats them as if they were tokens, sitting in the sequence alongside the text.

That is the whole trick. Once the projector is trained, the language model attends to image patches exactly as it attends to words, which is why vision capability can be added to an existing model without rebuilding it.

Images cost tokens, and more than you expect

Because patches become tokens, images consume context. The arithmetic varies by model, but the shape is consistent: an image at moderate resolution is several hundred tokens, and a high-resolution image processed with tiling can be well over a thousand.

Models that support dynamic resolution — Qwen-VL's approach, for instance — split a large image into tiles and process each, so a full-page scan can cost as much as several pages of text. Two consequences:

- **Context fills fast.** Six document photographs can consume most of an 8k window before the question is asked.

- **Cost is not intuitive.** On a rented endpoint, an image-heavy request can be more expensive than a long text one, and the price list only mentions tokens.

If a model exposes a resolution or tile limit, it is a real dial. Halving the resolution on a task that does not need fine detail can quarter the token cost with no measurable quality change, and you should test that rather than assume either way.

The families

Qwen-VL has the widest size range and the strongest document and OCR results among the open models. **Gemma 3** adds vision from 4B upwards, which makes it the smallest genuinely useful open vision model for a laptop. **Llama 3.2 Vision** covers 11B and 90B. **InternVL** is a strong research line with many sizes. **Pixtral** comes from Mistral. **Molmo**, from AI2, is notable for releasing its training data as well as its weights, which makes it the family to reach for if you need to defend how the model was built.

Serving support lags behind releases here, more than for text models. A new vision model may work in `transformers` weeks before llama.cpp or your preferred server supports it, and half-supported vision in a serving stack produces confusing results rather than clean errors. Check support before you plan around a specific model.

What they are good at, and where they fail

Good, and genuinely useful: describing an image, answering questions about a photograph, reading a screenshot, extracting fields from a reasonably clean document, transcribing handwriting at moderate quality, and describing a chart's overall shape.

The failures are consistent across the family and worth knowing before you build:

- **Counting.** Ask how many people are in a photograph and expect errors above about five. This is a known and persistent weakness.

- **Precise spatial relations.** Left of, above, third from the right — unreliable.
- **Dense document layout.** A table with merged cells, a multi-column form, a scan at an angle. The model produces plausible, well-formatted, wrong output, which is worse than an error.
- **Small text.** If a human squints, the model fails, and it will not tell you it failed.
- **Reading exact numbers from charts.** It reads the trend, not the values.

For document work specifically, a dedicated OCR pipeline — Tesseract, PaddleOCR, or a specialised document model — followed by a text model is often more accurate and far cheaper than a vision-language model doing both jobs. Test the boring pipeline before assuming the impressive one wins.

Prompting a vision model

Two habits that measurably help. Ask for the evidence: *quote the line that contains the invoice number* rather than *what is the invoice number*, which makes the failure visible when it happens. And ask for a schema, so a missing field comes back as null rather than as an invented value.

> *Audio input is following the same architectural path — an audio encoder, a projector, the same language model — and the same warning applies: capability arrives in the weights well before it arrives in the serving stacks most people use.*

BEFORE YOU MOVE ON

A team extracts fields from photographed invoices with a vision-language model. Accuracy is high on clean invoices and poor on dense multi-column ones, with confident wrong values rather than errors. What is the best next step?

A. Increase the context window, since dense documents need more room for the model to reason.

B. Lower the temperature to zero, which will stop the model inventing values.

C. Compare against a dedicated OCR step feeding a text model, and require the model to quote the source line for each field so wrong reads become visible.

D. Switch to a larger vision model, since dense layout understanding scales with parameters.

29 · 8 min

Formats, and what conversion costs you

THE ONE THING TO KEEP

Conversion between weight formats is where chat templates, stop tokens and unrecognised architectures get silently mangled, so prefer official conversions and verify a converted file against the original on the same prompts.

Five formats you will meet

safetensors is the canonical distribution format: a JSON header of tensor names, shapes and byte offsets, then the raw data. No code, memory-mappable, sharded across files for large models. This is what a repository publishes and what `transformers`, vLLM and most training code load.

GGUF is llama.cpp's single-file format. It holds the weights, the quantisation scheme, the tokenizer and the chat template in one file, which is why an Ollama or llama.cpp model is a single download with no dependencies. It is the format of local inference on CPUs and consumer hardware.

MLX is Apple's array framework format, for Apple Silicon. It uses unified memory well and is usually the fastest option on a Mac.

ONNX is a portable computation graph, used to run models in environments without PyTorch — browsers via WebAssembly, mobile runtimes, and various embedded stacks.

ExecuTorch and **TensorRT-LLM engines** are ahead-of-time compiled artefacts for phones and Nvidia servers respectively. They are built for one target and one configuration, and they are not portable.

There are also packed quantised checkpoints — GPTQ, AWQ, and bitsandbytes-quantised saves — which are safetensors files containing already-quantised weights plus scale tensors, loaded by a matching kernel.

Converting, in practice

The common direction is safetensors to GGUF:

```
git clone https://github.com/ggml-org/llama.cpp && cd llama.cpp
pip install -r requirements.txt

python convert_hf_to_gguf.py /path/to/model --outfile model-
f16.gguf --outtype f16
./llama-quantize model-f16.gguf model-Q4_K_M.gguf Q4_K_M
```

Two stages, deliberately: a lossless conversion to a 16-bit GGUF, then quantisation from it. Keep the intermediate if you have the disk, because producing a different quantisation later is then minutes rather than a re-conversion.

For Apple Silicon:

```
pip install mlx-lm
mlx_lm.convert --hf-path Qwen/Qwen3-8B -q --q-bits 4 --mlx-path
./qwen3-8b-mlx
```

What conversion breaks

Conversion is where silent damage enters the ecosystem, and it is worth knowing the three ways.

The chat template. GGUF stores the template in its metadata. If the conversion script does not understand a newer template, it can write a default one, and every user of that file then gets slightly wrong formatting with no indication. This is the most common defect in third-party GGUF uploads. Check it:

```
./llama-server -m model.gguf --verbose 2>&1 | grep -i "chat
template"
```

Special tokens and stop conditions. A model with several end-of-turn tokens can lose one in conversion, which produces generation that will not stop. The next lesson is about diagnosing exactly this.

Architecture support. A converter that does not recognise an architecture may map it onto the nearest one it knows. The file loads, produces text, and is subtly wrong. If the output is fluent but the model seems dumber than its reputation, this is a candidate.

The safe habit is to prefer conversions published by the model's own organisation, and where you use a third party, use one whose uploads are widely used and who states the source revision. When you convert yourself, verify: run the same five prompts through the original with `transformers` at temperature 0 and through the converted file at temperature 0, and read both. They will not be token-identical — different kernels, different arithmetic order — but they should be recognisably the same model.

Which format for which job

- Training or fine-tuning: safetensors, full precision.
- Serving on a GPU to many users: safetensors with vLLM, or an AWQ or fp8 quantised checkpoint.
- One person on a laptop or a CPU server: GGUF.
- A Mac: MLX first, GGUF second.

- A phone or a browser: ONNX or ExecuTorch, and expect to spend real time on it.

You will end up with more than one copy of the same model in different formats. That is normal and it is why the arithmetic on download sizes earlier in the course was worth doing.

BEFORE YOU MOVE ON

A third-party GGUF of a well-regarded model produces noticeably worse output than the same model served from safetensors, though it loads and generates fluently. What should you check first?

A. The quantisation level, since Q4 always costs a large amount of quality on instruction following.

B. The chat template and special tokens stored in the GGUF metadata, which conversion scripts can write incorrectly without any error.

C. The context length, since GGUF files default to a shorter window than the model supports.

D. The hardware, since CPU inference produces lower-quality output than GPU inference.

30 · 8 min

Why it will not stop talking

THE ONE THING TO KEEP

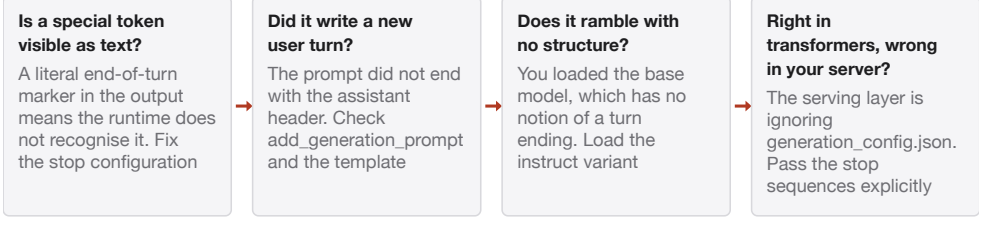
Runaway generation is almost always a stop condition rather than a bad model — several end-of-turn tokens, a template that never opened the assistant turn, a lossy conversion, or a serving layer ignoring generation_config.json.

A specific, common, fixable failure

The model answers your question, and then keeps going: another paragraph, then a fabricated user message, then an answer to that, until the token limit ends it. Or it stops correctly in one tool and rambles in another with the same weights.

This is not the model being poor. It is a stop condition that is missing, wrong, or not being applied, and there are four places to look.

Four checks, about thirty seconds each



One of them is nearly always the answer. Whatever the outcome, set max_tokens as well: it turns a runaway into a truncated answer rather than a bill, and a reasoning model looping inside its thinking block will emit tens of thousands of tokens without one.

1. The end-of-turn token

Generation stops when the model emits a token the runtime recognises as terminal. Which token that is lives in the model's configuration, and modern models often have more than one:

```
// generation_config.json, Llama 3.1
{
  "bos_token_id": 128000,
  "eos_token_id": [128001, 128008, 128009],
  "temperature": 0.6,
  "top_p": 0.9
}
```

Three end tokens: end-of-text, and two end-of-turn variants. If your runtime honours only the first, the model emits `<|eot_id|>` at the correct moment, the runtime does not recognise it, and generation continues. The text after that point is the model continuing a transcript, which is why it so often invents a new user turn.

The check is direct: log the raw token ids of a generation and see what it emitted where you wanted it to stop.

2. The template did not close the turn

If the prompt did not end with the assistant header — the `add_generation_prompt` problem from the template lesson — the model is not answering, it is continuing a document. Documents do not end after one reply.

3. Conversion or serving lost a token

A GGUF conversion that failed to carry across one of several end tokens produces exactly this symptom, and it is the most frequent defect in third-party conversions. Similarly, some serving layers apply `generation_config.json` and some ignore it entirely, using only `eos_token_id` from `config.json`.

Belt and braces, and worth doing on any model you did not convert yourself:

```
# llama.cpp: name the stop strings explicitly
llama-server -m model.gguf --chat-template llama3

# OpenAI-compatible clients: pass stop sequences
{"model": "...", "messages": [...], "stop": ["<|eot_id|>", "
<|im_end|>"], "max_tokens": 512}
```

4. It is a base model

Base models have no notion of a turn ending. They continue until the limit. If you loaded `-Base` by accident, everything above is irrelevant and the fix is to load the instruct variant.

Stop strings, and their sharp edge

Most runtimes let you supply stop strings that halt generation when they appear in the output. They are matched on the decoded text, so they are useful for formats — stopping at a closing brace, at a section marker, at `\nUser:`.

Two cautions. The stop string is normally excluded from the output, so a stop at `}` may cut the brace you wanted. And a stop string that can appear inside legitimate output truncates good answers: stopping at a newline is a classic way to break every multi-line reply.

Always set a limit

Whatever else you do, set `max_tokens`. It is the backstop that turns a runaway into a truncated answer rather than a bill or a hung request. Reasoning models make this essential rather than prudent: a model that loops inside its thinking block can emit tens of thousands of tokens on a question it will never answer, and without a cap it will.

A reasonable default is twice the longest legitimate answer you expect. If you routinely hit the cap, that is data — either the task needs more room or something is looping.

The two-minute diagnosis

1. Does the output contain a special token as visible text, such as a literal `<|eot_id|>`? The runtime is not recognising it. Fix the stop configuration.
2. Does it write a new user turn? Check `add_generation_prompt` and the template.

3. Does it ramble without structure? Check whether you loaded a base model.
4. Does it stop correctly in `transformers` and not in your server? The serving layer is ignoring `generation_config.json`. Pass stops explicitly.

Four checks, each about thirty seconds, and one of them is nearly always the answer.

BEFORE YOU MOVE ON

A model stops correctly when run through transformers but rambles when served through a local OpenAI-compatible server, with a literal <|eot_id|> appearing in the output text. What does that tell you?

A. The server is generating past the model's context window, so tokens are being decoded from corrupted state.
B. The model file is corrupt, since a valid model would never emit a special token as text.
C. The server is not treating that token as terminal, so the model's correct stop signal is being decoded as ordinary text and generation continues.
D. The chat template is wrong, since a correct template would prevent the token appearing in the output.

CASE STUDY · MODULE 3

The model that would not stop talking, two days before the batch

A coaching institute in Kota running a pilot doubt-solving assistant for four hundred first-year engineering-entrance students, with the batch starting on Monday.

The assistant had worked for six weeks. A student typed a physics doubt, the model answered in four or five lines, and a teacher reviewed anything flagged. It ran in a notebook on a rented GPU during development. On Thursday the institute's system administrator, Farhan, moved it to the machine it would actually live on: a desktop with a 12 GB card under the server-room air conditioner, running a GGUF file through a local server so the pilot would not depend on the institute's unreliable link.

On Thursday evening it stopped stopping.

A student asked why a block on an incline does not slide. The model answered correctly in four lines. Then it wrote another paragraph. Then it wrote `Student:` and invented a follow-up question, answered that, invented another, and continued until it hit the token limit five hundred tokens later. Every answer was like this. In the notebook, the same model, the same prompt, the same question had stopped cleanly.

What Farhan checked, in order

He had two days and he started with the cheapest checks.

Was a special token visible in the output? He turned off the server's own formatting and looked at the raw text. There it was: a literal end-of-turn marker sitting in the middle of the reply, printed as characters rather than acted on. The model had said it was finished at exactly the right place. The runtime had not noticed.

Why had the runtime not noticed? He compared the two files. The original repository's `generation_config.json` listed three end-token ids: an end-

of-text token and two end-of-turn variants. The GGUF he had downloaded — a third-party conversion, five thousand downloads, no card beyond a line saying which quantisation it was — carried one. The other two had not survived the conversion.

Was anything else wrong with the file? This was the question he could not answer quickly. The same conversion that dropped two stop tokens writes the chat template into the file's metadata, and a converter that does not understand a newer template writes a default one instead. The server's startup log printed the template it had loaded. Farhan could read it, and he could see it was *a* template. Whether it was *the* template the model had been trained on was a comparison he had no reference for, because he did not have the original repository on that machine.

Friday morning, with the batch on Monday

Two routes, and they cost different things.

Patch it. Pass the missing stop strings explicitly in every request, and set `max_tokens` to four hundred as a backstop. Twenty minutes of work. The runaway stops immediately. The risk is that the template may also be subtly wrong, which would not look like a bug — it would look like the model being a bit worse at physics than it was in October, and nobody would be able to say when it had changed.

Convert it themselves. Download the original 16 GB of safetensors, convert to a 16-bit GGUF, quantise to the same bit width, verify the template and the stop tokens against the source, and run the same five prompts through both at temperature zero to confirm they are recognisably the same model. On the institute's connection the download alone was an overnight job, the conversion and quantisation about forty minutes, and the verification an hour. Friday night and most of Saturday, during an admissions week in which Farhan was also the person who fixed the biometric attendance machine.

The head of the institute asked the only question that mattered: if we patch it and something is wrong with the template, when do we find out? Farhan's honest answer was that they would find out from students saying the assistant

had got worse, which is the worst kind of bug report, because by then four hundred people have a view about the tool.

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

Farhan did both, in that order. The stop strings and a four-hundred-token cap went in on Friday morning and the runaway ended inside the hour, which meant the pilot was safe whatever else happened. The download ran overnight, the conversion ran on Saturday, and the verification found that the template in the third-party file was in fact correct — the conversion had lost stop tokens and nothing else. He kept the 16-bit intermediate GGUF on the desktop's second drive, so producing a different quantisation later became a two-minute job rather than another overnight download. The institute's run-book gained four lines that now run before any model reaches a student: print the chat template the server loaded, print the end-token ids, run five fixed prompts at temperature zero against the original, and set max_tokens always.

The same weights stopped correctly in the notebook and ran away on the server. Name the two places that difference can come from, and say how you would tell them apart.

Either the file changed — a conversion that carried fewer end-of-turn tokens than the original `generation_config.json` lists — or the serving layer changed, because some servers read `generation_config.json` and some use only `eos_token_id` from `config.json`. You tell them apart by looking at the raw output: a special token appearing as visible text means the model emitted the right token and the runtime did not recognise it, which points at the file or the server's stop configuration. If the output instead writes a new user turn with no special token, the prompt did not end with the assistant header and the problem is the template.

Farhan could see that a chat template had been loaded but not whether it was the right one. Why is a wrong template harder to

deal with than a missing stop token?

Because it produces no error and no obvious symptom. A missing stop token gives you a runaway, which is loud. A wrong template puts the model slightly off the distribution it was trained on, so quality drops without anything breaking, and the drop is attributed to the model rather than to the file. The only way to detect it is comparison: run the same prompts through the original weights and the converted file at temperature zero and read both. They will not be token-identical, because kernels and arithmetic order differ, but they should be recognisably the same model.

He kept the 16-bit intermediate GGUF rather than deleting it after quantising. What does that buy, and what does it cost?

It buys the ability to produce any other quantisation in minutes rather than repeating a 16 GB download and a conversion — useful when a bit width turns out to be too aggressive for long-context or non-English work, which is exactly where quantisation damage appears first. It costs disk: roughly sixteen gigabytes for an 8B model. On a desktop with a second drive that is a trivial price, and it is the same reasoning as keeping a copy of the original weights so that a repository being renamed or withdrawn is not your problem.

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 In a Llama-3.1-8B configuration, hidden size is 4096 and intermediate size is 14336. Which part of each layer holds most of the parameters?

A. The normalisation layers, which scale with hidden size at every position

B. The attention projections, since attention is what the architecture is named for

C. The feed-forward matrices, which are roughly eighty per cent of a layer

D. The embedding table, which dominates every layer's parameter count

2 Two models have identical parameter counts, layer counts and head dimensions. One has 32 key-value heads, the other 8. What differs for somebody running a server?

A. The second generates tokens about four times faster than the first

B. The second needs a quarter of the KV cache, so one card holds far more users

C. The first supports longer context because it stores more positional detail per token

D. Nothing operationally; head grouping only affects stability during training

3 A local model answers the question, then writes a new user message and answers that one too. Which check is most likely to explain it?

A. Whether the prompt ended at the user's turn instead of the assistant header
B. Whether the tokenizer's vocabulary matches the weights that were loaded
C. Whether the context limit is long enough to hold the whole conversation history
D. Whether the temperature is too high for this kind of task

4 A team formats messages with the chat template as a string, then tokenises that string with the library's default settings. What has quietly gone wrong?

A. The template was applied twice, duplicating the whole system turn
B. Tokenising a formatted string loses the leading spaces, so every word becomes a new token
C. The special tokens were stripped, so the model sees plain text with no role markers
D. A second beginning-of-sequence token was added at the start

5 A model advertises 128k of context and passes a single-needle retrieval test at that length. What does that result support?

A. That the cache at that length is smaller than the model's own weights
B. That the model reasons reliably over a whole 128k document
C. That it can find one distinctive sentence, the easiest long task
D. That prefill at 128k costs about sixteen times what prefill at 8k costs

6 The same GGUF file stops correctly in one runtime and rambles in another, inventing user turns. Where do you look first?

A. The quantisation, since four-bit files lose precision on the end-of-turn token
B. Whether the runtime recognises all of the model's end-of-turn token ids
C. The temperature, because greedy decoding never emits a stop token
D. The random seed, since a different seed can miss the terminal token entirely

MODULE 4

Making it run on the machine you have

Most writing about local models assumes a graphics card that most readers do not own. This block does the arithmetic instead: what quantisation actually does to the numbers, which bit width to pick, how to predict tokens per second from memory bandwidth before you download anything, how to serve many users on one GPU, where to get compute for nothing, when renting stops being cheaper than owning, and what genuinely runs on a phone.

> BY THE END OF THIS MODULE YOU CAN
>
> Size, quantise and serve an open model on hardware you actually have — a laptop, a phone, a free cloud tier or a rented GPU — predict its generation speed from memory bandwidth before running it, choose a bit width you can defend, and state the monthly token volume at which renting stops being cheaper than owning

31 · 9 min

Choosing a Model for the Job

THE ONE THING TO KEEP

Fit the constraints first, memory and latency and licence and language, then test the two or three models left standing.

Write the constraints before you look at candidates

Five numbers, on paper, before any browsing:

1. Memory budget on the machine that will actually run it.
2. Latency budget at p95, from a real user's point of view.
3. Requests per day.
4. Licence constraint (commercial, redistributable, attribution acceptable).
5. Whether data may leave the building.

Most shortlists collapse to two or three candidates immediately. That is the point.

The memory arithmetic

Weights = parameters × bytes per weight.

- bf16 ≈ 2 bytes → an 8B model is about 16 GB
- Q8 ≈ 1 byte → about 8.5 GB
- Q4_K_M ≈ 0.6 bytes → about 4.9 GB

KV cache = 2 × layers × kv_heads × head_dim × bytes, per token. For Llama-3.1-8B (32 layers, 8 KV heads, head dim 128, fp16): 2 × 32 × 8 × 128 × 2 = 131,072 bytes, or 128 KB per token.

- 8k context → about 1.0 GB
- 32k context → about 4.2 GB

That arithmetic is why a model that loads happily at 2k context dies at 32k. Add roughly 1 GB of runtime overhead, and aim for weights + KV + 1 GB under 90% of your budget.

If you have 8 GB of RAM and no discrete GPU

This is most people, and most writing about local models pretends otherwise. Honestly:

- **3B to 4B instruct at Q4.** Comfortable. About 2.5 GB of weights, room for 8k context, roughly 8 to 20 tokens per second on a recent CPU. Genuinely good at extraction, classification, tagging, rewriting, short summaries.
- **7B to 8B at Q4.** Fits at about 4.9 GB, but leaves little for context and nothing for your browser. Expect 4 to 10 tokens per second. Fine for overnight batch work, painful for chat.
- **14B and above.** Not on that machine. Not "slow" — not usable.

Apple Silicon changes the maths, because unified memory is shared with the GPU and bandwidth is high. A 16 GB M-series laptop behaves roughly like a 14B-at-Q4 machine, which is a real capability difference from a 16 GB x86 laptop with integrated graphics.

On mixture-of-experts: a 30B model with 3B active still needs the whole 30B resident — around 18 GB at Q4 — but computes at roughly 3B speed. Excellent if you have the RAM. Irrelevant if you do not.

And renting is not cheating. OpenRouter, Together, Fireworks, DeepInfra and others serve these same open weights per token. The freedom of open weights is that you *could* run it yourself and nobody can take the model away; it does not oblige you to buy a GPU. Self-host when data cannot leave, when volume makes it cheaper, or when you need it offline.

Match the shape of the task

- **Extraction, classification, routing.** Smallest instruct model that passes. Use JSON schema or grammar-constrained decoding, temperature 0.

Model choice matters less than constraints here.

- **Long documents.** Advertised context is not usable context. Most models degrade well before their maximum. Test retrieval from the middle of your own longest document before believing the number on the card.
- **Code.** Use the coder-tuned variant and check which languages the card names. The gap between a general and a coder model of the same size is large.
- **Tool use and agents.** Check the card explicitly claims function-calling training, then test with your real schemas. This capability varies far more between models than chat quality does.
- **Non-English.** Check the language list, then test. Also check tokenizer efficiency: some models spend two or three times as many tokens on non-Latin scripts, which costs you money and eats your effective context at the same time.

The shortlist rule

Three candidates. No more than one size class apart. All licence-clear. Then stop reading and start measuring, which is the next lesson.

BEFORE YOU MOVE ON

On a 16 GB laptop with no GPU, a 14B model at Q4 (about 8 GB of weights) loads fine but the machine thrashes once 32k context is enabled. Why?

A. The KV cache grows with context length and can reach several gigabytes at 32k, on top of the weights and everything else the OS is holding.

B. Quantized loading briefly materialises the fp16 weights in memory, so a Q4 model still needs its full-precision footprint at startup.

C. 14B models require a GPU, and CPU inference is not actually supported at that size.

D. Longer context enlarges the embedding and vocabulary tables the model must keep resident.

32 · 10 min

What quantisation actually does

THE ONE THING TO KEEP

Quantisation stores weights as small integers with a shared scale per block, so a 4-bit model is really about 5 bits per weight, and because larger models carry more redundancy, a bigger model quantised harder usually beats a smaller one at full precision for the same memory.

Storing a number in four bits

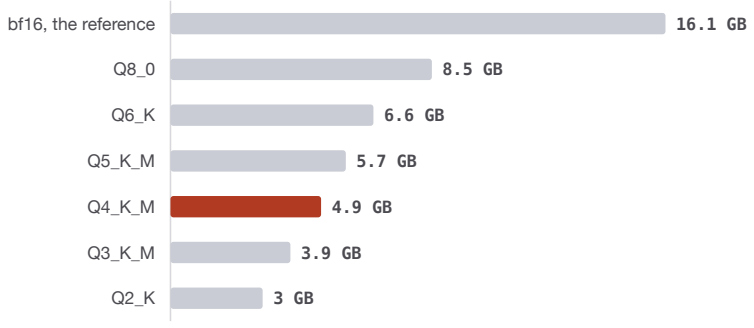
A weight in a released model is usually bfloat16: sixteen bits, of which eight are exponent. Quantisation stores it in fewer bits, and the whole art is in how you keep the information that matters.

Naively rounding every weight to one of sixteen values across the model's full range would be a disaster, because a handful of large weights would stretch the range and everything else would collapse to zero. So quantisation is done **block-wise**. Split the weights into blocks of 32 or 64. For each block, find the largest magnitude, store a scale factor, and store each weight as a small integer index into that block's range.

```
block of 32 weights, 4-bit:
  32 indices x 4 bits           = 128 bits
  1 scale in fp16               =  16 bits
  (some schemes also a minimum) =  16 bits
  ----------------------------------------
  160 bits / 32 weights          = 5.0 bits per weight
```

This is why a model described as 4-bit is not exactly half the size of an 8-bit one. Real formats land between 4.5 and 5.5 bits per weight once the scales are counted, which is exactly why an 8B model at Q4_K_M is about 4.9 GB rather than 4.0 GB. If you have ever wondered where that extra gigabyte came from, it is the scales.

An 8B model on disk, by quantisation



Q4_K_M is about 4.9 GB rather than 4.0 because each block of weights carries a scale, and often a minimum, alongside the indices — roughly five bits per weight, not four. That extra gigabyte is the scales, and it is the answer to where it came from.

The K in K-quants

llama.cpp's K-quant family does something smarter than uniform treatment: it assigns different precision to different tensors. The layers that hurt most when degraded — in practice, parts of the attention and one of the feed-forward matrices — keep more bits, while the rest take fewer. The suffixes encode this. `Q4_K_S` is the small variant, `Q4_K_M` the medium, and the M version spends its extra half-bit where measurements said it mattered.

More recent work adds an **importance matrix**: run a calibration corpus through the model, record which weights have the largest effect on the output, and let the quantiser protect them. Files named with `imatrix` were built this way, and the benefit is largest at very low bit widths, where there is least room for error.

The other methods, and what each assumes

Round to nearest is the baseline: no calibration, instant, and the weakest.

GPTQ processes weights layer by layer, using a small calibration dataset and second-order information to compensate the remaining weights for each rounding error as it goes. Slow to produce, good results, GPU-oriented.

AWQ starts from an observation: roughly 1% of weight channels matter far more than the rest, and which ones can be identified from the *activations*

rather than the weights. Protect those channels by scaling, quantise the rest hard. Fast and strong, and widely used for GPU serving.

bitsandbytes NF4 uses a 4-bit format shaped for normally distributed values, which weights approximately are. It is the format under QLoRA fine-tuning, and it is designed to be loaded and trained against rather than to be maximally compact.

FP8 is not really quantisation in the same sense — it is a hardware-supported floating point format on recent Nvidia cards, giving near-lossless quality at half the memory with no calibration. Where the hardware supports it, it is the easy win.

Why activation outliers matter

The reason naive 8-bit integer quantisation failed on transformers, and the reason the methods above exist, is a specific empirical fact: in large models a few hidden dimensions carry activations with magnitudes hundreds of times larger than the rest. Quantise a matrix multiplication that includes those channels uniformly and the small values disappear entirely.

Every serious method handles this somehow — by keeping outlier channels in higher precision, by scaling them before quantising, or by moving the difficulty from activations into weights. It is worth knowing because it explains why quantisation quality is not a smooth function of bit width and why methods that ignore the phenomenon fail suddenly rather than gradually.

What it costs, in one sentence you can act on

For models above about 7 billion parameters, going from 16-bit to a good 4-bit scheme typically costs a small fraction of a point on most tasks — usually invisible against the noise of a 40-example evaluation set.

Which produces the rule that decides most local setups: **a larger model quantised harder usually beats a smaller model at full precision**, for the same memory. A 13B at 4 bits and an 7B at 8 bits both occupy about 8 GB, and the 13B is generally better.

The rule has limits, and they matter. Below about 3 billion parameters the model has less redundancy to spare and quantisation hurts more. Quantisation degrades long-context retrieval and non-English performance more than it degrades short English chat, which is precisely the sort of thing a general benchmark hides. And below 4 bits the curve steepens sharply. Those limits are the subject of the next lesson.

BEFORE YOU MOVE ON

Why is an 8B model in Q4_K_M about 4.9 GB rather than the 4.0 GB that four bits per weight would predict?

A. The tokenizer and configuration files add roughly a gigabyte to the download.

B. Quantisation stores per-block scale factors alongside the indices, and K-quants keep selected tensors at higher precision, which together push the effective rate to about five bits per weight.

C. The embedding matrices cannot be quantised and remain at 16 bits.

D. GGUF files include the KV cache allocation in the file size.

---

33 · 8 min

Choosing a bit width you can defend

THE ONE THING TO KEEP

Spend memory on parameters before precision down to about four bits, but expect quantisation to damage long-context retrieval and non-English performance well before it damages short English chat, so test at the bit width you will ship.

The ladder, and what each rung costs

Ordered from least to most compressed, with the honest character of each:

- **bf16 / fp16** — the reference. Everything else is measured against this.

- **fp8 or Q8_0** — effectively indistinguishable from the reference on ordinary tasks. Use it when memory allows; there is nothing to think about.
- **Q6_K** — very close to Q8 at three-quarters of the size. An underused sweet spot for people with a little headroom.
- **Q5_K_M** — a small, usually unnoticed loss. Sensible when Q4 makes you nervous and Q6 does not fit.
- **Q4_K_M** — the community default, and correctly so. The knee of the curve for models above about 7B.
- **Q3_K_M** — noticeably degraded. Instruction adherence weakens first; the model starts ignoring parts of a multi-clause request.
- **Q2_K** — usually a mistake. A smaller model at Q4 will almost always be better.

The rule that follows

Spend your memory on parameters before you spend it on precision, down to about 4 bits.

Given 8 GB to work with, a 13B at Q4 beats an 8B at Q6 beats a 7B at Q8, on most tasks, most of the time. Given 5 GB, an 8B at Q4 beats a 4B at Q8.

The rule inverts below 4 bits and at small sizes, and both exceptions have the same cause. Quantisation works because large models carry redundancy — many weights contribute a little, so small errors average out. A 1.5B model has less of that redundancy per parameter, so the same bit reduction costs it more. Below about 3B, prefer Q6 or Q8 and accept the smaller model.

What degrades first, and where the benchmarks hide it

The damage is not spread evenly across capabilities, and a general benchmark average conceals this. In rough order of sensitivity:

1. **Long-context retrieval.** Finding a specific fact deep in a long document depends on fine distinctions between key vectors, which is exactly what coarse quantisation blurs.

2. **Non-English performance.** Lower-resource languages sit in less well-populated regions of the model's representation, with less redundancy supporting them. A quantised model can lose noticeably more of its Hindi than its English.
3. **Multi-step arithmetic and code.** Errors compound; a small increase in per-step error is visible at the end of a chain.
4. **Instruction adherence over long outputs.** The model complies for two paragraphs and drifts.
5. **Short English chat.** The last thing to break, and unfortunately the first thing everyone tests.

If any of the first four describe your use, test at your quantisation rather than trusting a published perplexity number.

Perplexity is a weak proxy

Most quantisation comparisons report perplexity on a text corpus. It is cheap, it correlates with quality, and it is not sensitive enough for this decision: two quantisations can be within a hundredth of a perplexity point and behave differently on structured output.

The better measurement, covered in more detail later, is agreement with the full-precision model. Run both on 200 prompts and compare their next-token distributions, or more simply compare their actual answers on your own examples. A quantisation that agrees with the reference on 96% of your task is a different proposition from one that agrees on 78%, and perplexity may not separate them.

A practical procedure

1. Compute how much memory you have for weights, after subtracting KV cache and about 1 to 2 GB of overhead.
2. Pick the **largest model** whose Q4_K_M fits.
3. If it fits with room to spare, move up to Q5 or Q6 rather than to a bigger model, unless the next size up also fits at Q4.
4. Run your 40 examples at the chosen quantisation and at one step higher.

5. If the difference is inside the noise, keep the smaller file. If it is not, you have learned something specific to your task that no general table would have told you.

Step 4 costs an hour and settles arguments that otherwise recur for months.

> *On downloading: repositories from prolific quantisers such as bartowski and the official organisation uploads generally publish the whole ladder with the importance-matrix variants. Take Q4_K_M first, and take one neighbour so you can run step 4 without a second download.*

BEFORE YOU MOVE ON

A team runs a Q4_K_M 8B model for Bengali summarisation and finds quality clearly worse than the fp16 version, although published perplexity and English benchmark differences were negligible. What is the most likely reason?

A. Quantisation is unreliable for any model below 13B, so the 8B is simply too small to quantise.

B. The GGUF conversion lost the Bengali tokens from the vocabulary.

C. Lower-resource languages rely on less redundant regions of the model, so they lose more to quantisation than English does, and English-weighted benchmarks do not show it.

D. Perplexity was measured on the wrong corpus, so the published numbers are invalid.

34 · 10 min

llama.cpp, end to end

THE ONE THING TO KEEP

llama.cpp is the engine under most desktop AI tools, and its important dials are the number of layers offloaded to the GPU, the context length that sets your cache allocation, and llama-bench reporting prefill and decode speed as two separate numbers.

The tool that made local models real

llama.cpp is a C++ inference engine with no Python dependency, support for CPU, Nvidia, AMD, Apple Metal and Vulkan, and a single-file model format. Ollama, LM Studio, Jan and most desktop AI applications are wrappers around it. Learning the underlying tool takes an hour and gives you every dial the wrappers hide.

Installing

```
# macOS
brew install llama.cpp

# Debian or Ubuntu, prebuilt
# see the project releases; or build with the backend you need:
git clone https://github.com/ggml-org/llama.cpp && cd llama.cpp
cmake -B build -DGGML_CUDA=ON        # Nvidia; use -
DGGML_METAL=ON on a Mac
cmake --build build --config Release -j
```

The build flag is the important part. A CPU-only build on a machine with a GPU works and is several times slower, and it is a common quiet mistake.

Three binaries

`llama-cli` runs one prompt. `llama-server` exposes an OpenAI-compatible HTTP endpoint and a small web interface. `llama-bench` measures speed. You will use the server for almost everything.

```
llama-server   -hf bartowski/Qwen2.5-7B-Instruct-GGUF:Q4_K_M
-c 8192   -ngl 99   --flash-attn   --port 8080
```

`-hf` downloads directly from Hugging Face with the quantisation after the colon, which removes a manual download step.

The flags that matter

`-ngl N` — number of layers to offload to the GPU. This is the single most important flag on a machine with a graphics card. `-ngl 99` means all of them. If the model does not fit in VRAM, reduce it: the layers you offload run on the GPU, the rest on the CPU, and speed scales roughly with the fraction offloaded. Half the layers on a GPU is much better than none.

`-c N` — context length. It sets your KV cache allocation, so it is the flag that decides whether the model fits. Do not set it to the model's maximum out of habit.

`--flash-attn` — a more memory-efficient attention implementation. Turn it on; there is rarely a reason not to.

`--cache-type-k q8_0 --cache-type-v q8_0` — quantise the KV cache, roughly halving its memory. Test the quality effect on long-context tasks.

`-t N` — CPU threads. The default is usually right; setting it above your physical core count reliably makes things slower.

`--parallel N` and **`-np N`** — number of concurrent sequences the server will handle. The context you set is divided among them, so `-c 8192 --parallel 4` gives each slot 2048 tokens.

`--mlock` — lock the weights in RAM so the operating system cannot page them out. Worth it on a machine under memory pressure, and it will fail with-

out the right permissions.

`--no-mmap` — turns off memory mapping. Generally leave mapping on; it lets a model start before the whole file is read.

Measuring before you tune

```
llama-bench -m model-Q4_K_M.gguf -p 512 -n 128 -ngl 99
```

This reports two numbers separately, and the separation is the point. **Prompt processing** is prefill: how fast it reads your input, measured in tokens per second and usually in the hundreds or thousands. **Text generation** is decode: how fast it produces output, usually one or two orders of magnitude slower. The next lesson explains why they differ so much.

Run this before and after any change. It is the difference between knowing that `--flash-attn` helped and believing it did.

Serving it

The server speaks the OpenAI API, so every client works:

```
curl localhost:8080/v1/chat/completions -H "Content-Type: ap-
plication/json" -d '{
  "messages": [{"role":"user","content":"Explain a hash join in
three sentences."}],
  "temperature": 0, "max_tokens": 200
}'
```

The template comes from the GGUF metadata automatically. If output looks wrong, check what template it loaded — the startup log prints it, and a wrong template here is the most common cause of disappointing local results.

When to leave

llama.cpp is the right tool for one user, a handful of users, CPU inference, Apple hardware, and anything that must run offline on a laptop. It is not the

right tool for serving thirty concurrent users on a datacentre GPU, where continuous batching and paged attention give several times the throughput. That is vLLM, two lessons from now.

BEFORE YOU MOVE ON

A 7B model at Q4 needs about 4.9 GB but the GPU has only 4 GB free. What does -ngl let you do?

A. Nothing useful; the model must fit entirely in VRAM or run entirely on the CPU.

B. Offload as many layers as fit and run the rest on the CPU, giving speed between the two extremes roughly in proportion to the fraction offloaded.

C. Compress the model further at load time so that it fits in the available VRAM.

D. Swap layers in and out of VRAM per token, so the whole model effectively runs on the GPU.

---

35 · 8 min

Ollama and the friendly path

THE ONE THING TO KEEP

Wrappers like Ollama choose your quantisation, sampling defaults and context length for you, and the historic 2048-token context default is the single most common cause of a local model appearing to ignore the start of a document.

Two commands to a running model

```
curl -fsSL https://ollama.com/install.sh | sh   # or brew install ollama
ollama run gemma3:4b
```

That is a working local model with a chat prompt, a downloaded and correctly templated file, and an HTTP API on port 11434. For a first hour it is unbeatable, and for many people it is the permanent answer.

The alternatives in the same category: **LM Studio**, a graphical application with a model browser and a server tab, free but not open source; **Jan**, open source and similar in shape; **GPT4All**, aimed at document chat on ordinary laptops. All of them wrap llama.cpp, so the underlying behaviour is what the previous lesson described.

The default that catches everybody

Ollama's default context length has historically been 2048 tokens, regardless of what the model supports. Send a longer prompt and the beginning is silently dropped.

The symptom is bizarre and very common: a model that answers well about the end of your document and appears not to have read the beginning, or a chat that forgets what was said ten turns ago. People report this as the model being poor. It is a configuration default.

```
# per session
ollama run qwen3:8b
>>> /set parameter num_ctx 32768

# or permanently, with a Modelfile
cat > Modelfile <<'EOF'
FROM qwen3:8b
PARAMETER num_ctx 32768
PARAMETER temperature 0
EOF
ollama create qwen3-long -f Modelfile
```

Check what you actually have with `ollama show qwen3:8b`, which prints the parameters and the template.

What else the wrapper decides for you

Quantisation. A tag such as `gemma3:4b` resolves to a specific quantisation, usually Q4_K_M. If you want Q6 or Q8 you must ask: `gemma3:4b-instruct-q8_0`. Run `ollama list` after pulling to see the size and confirm which you got.

Sampling defaults. Temperature, top-p and repetition penalty are set by the model's Modelfile, not by you, unless you override them. If you are comparing two models and did not set these explicitly, you are partly comparing their defaults.

The chat template. Handled for you, from the file's metadata, which is the main thing you are buying.

Model residency. Models unload after a few minutes of inactivity, so the first request after a pause pays the whole load time again. `OLLAMA_KEEP_ALIVE=-1` keeps a model resident indefinitely, at the cost of the memory.

Using it as a server

```
export OLLAMA_HOST=0.0.0.0:11434     # only on a network you
trust
curl localhost:11434/v1/chat/completions -d '{
  "model":"qwen3:8b",
  "messages":[{"role":"user","content":"hello"}],
  "temperature":0
}'
```

The OpenAI-compatible endpoint means every library and editor plugin works against it unchanged. That compatibility is worth more than any feature, because it means your evaluation script does not care whether it is talking to Ollama, llama.cpp, vLLM or a rented endpoint.

There is no authentication. Do not expose that port beyond a trusted network; a later lesson covers what happens when people do.

Importing your own file

```
cat > Modelfile <<'EOF'
FROM ./my-finetune-Q4_K_M.gguf
TEMPLATE "{{ .Prompt }}"
PARAMETER stop "<|im_end|>"
EOF
ollama create my-finetune -f Modelfile
```

Useful after you have converted your own fine-tune, and the point at which the wrapper stops being a black box.

When to move on

Stay with the friendly path while you are one person on one machine. Move to llama.cpp directly when you need a flag it does not expose, and to vLLM when you need to serve concurrent users properly.

The one habit to carry across regardless: **record the context length, the quantisation and the sampling parameters** alongside any result you intend to act on. Most irreproducible local benchmarks are irreproducible because one of those three was a default nobody wrote down.

BEFORE YOU MOVE ON

A user pastes a 6,000-word document into a local chat and the model answers only about the final section, seeming not to have read the rest. Nothing errors. What is the first thing to check?

A. The quantisation level, since a Q4 model may be dropping detail from long inputs.

B. Whether the runtime's context length is set well below the model's maximum, so the earlier tokens are being truncated silently.

C. The chat template, since a wrong template would drop the earlier turns.

D. The system prompt, which may be instructing the model to summarise only the conclusion.

36 · 9 min

Predicting speed before you download

THE ONE THING TO KEEP

Decoding reads every active weight from memory once per token, so tokens per second is bounded by memory bandwidth divided by model file size, while prefill is compute-bound and therefore one or two orders of magnitude faster.

Generation is limited by memory bandwidth, not by arithmetic

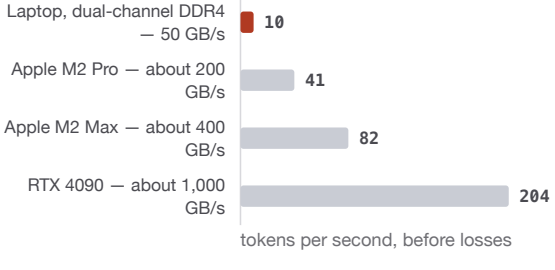
To produce one token, the machine must read every active weight from memory once. A 4.9 GB model means 4.9 GB moved per token. The arithmetic performed on those weights is trivial by comparison — modern processors can do far more multiplication than they can feed themselves data.

So the ceiling is:

```
tokens per second  =  memory bandwidth  /  bytes of active
weights
```

That one line predicts local model speed better than any benchmark chart, and you can evaluate it before downloading anything.

Ceiling for an 8B at Q4: bandwidth divided by 4.9 GB



Expect sixty to eighty per cent of these in practice. The same arithmetic explains why a 3B at 2.0 GB feels usable on the machine where the 8B feels broken, and it is the quickest check on anybody's claim about local speed: divide the bandwidth by the file size and ask what machine provides it.

Worked examples

An ordinary laptop, dual-channel DDR4-3200. Bandwidth around 50 GB/s. An 8B model at Q4 is 4.9 GB. The ceiling is 50 / 4.9 ≈ **10 tokens per second**, and in practice you will see 6 to 8. That is about four words per second: readable, not comfortable.

The same laptop, a 3B model at Q4 — 2.0 GB. Ceiling 25 tokens per second, real perhaps 15 to 18. This is the reason 3B models feel usable on machines where 8B models feel broken, and it is entirely about the file size.

Apple M-series with unified memory. An M2 Pro has around 200 GB/s, an M2 Max around 400, an M3 Max around 400. At 200 GB/s the 8B at Q4 gives a ceiling of 40 tokens per second, real around 25 to 30. The GPU and the CPU share this memory, which is why Apple laptops punch far above x86 laptops of the same nominal specification.

A consumer GPU. An RTX 3090 or 4090 has around 1,000 GB/s. Ceiling for the 8B at Q4 is roughly 200 tokens per second; real numbers of 100 to 140 are typical.

A datacentre GPU. An H100 has around 3,350 GB/s. The ceiling is enormous, which is why these machines are used for many users at once rather than for one.

Expect 60% to 80% of the theoretical number. The gap is attention over the growing cache, kernel overhead, and the fact that nothing achieves peak bandwidth.

Prefill is a different animal

Reading your prompt is compute-bound, not bandwidth-bound: all the tokens of the prompt go through the weights together, so the weights are read once for the whole batch. The cost is roughly 2 × parameters × tokens floating point operations, and processors have a great deal of arithmetic capacity.

This is why `llama-bench` reports hundreds or thousands of tokens per second for prompt processing and ten or forty for generation, on the same ma-

chine and model. It is not an inconsistency; the two phases are limited by different resources.

Two practical consequences. A long prompt with a short answer is fast. A short prompt with a long answer is slow. And on a CPU, prefill of a very long document can still take a while, because CPU arithmetic capacity is modest too.

Three things this explains at once

Why batching is nearly free. Serving eight users at once reads the weights once for all eight. Total throughput rises almost eight-fold while each individual user sees little slowdown. This is the entire economic basis of hosted inference.

Why mixture-of-experts models are fast. Only the active experts are read per token, so the bandwidth cost matches the active parameter count even though the memory occupied matches the total.

Why quantising speeds things up. Halving the bytes halves the reading, so it roughly doubles the speed. Quantisation is a speed optimisation as much as a memory one, which is often forgotten.

Use it as a purchase test

Before spending money, ask what bandwidth you are buying rather than what compute.

A machine with a large amount of slow memory will run big models slowly; a machine with fast memory will run models that fit quickly. For local inference specifically, memory bandwidth and memory capacity are the two numbers that matter, and raw compute barely enters into it.

> *The quickest sanity check on any claim about local speed: divide the claimed bandwidth by the model file size. If someone reports 60 tokens per second for a 5 GB model, they are claiming 300 GB/s of bandwidth, and you can ask what machine provides it.*

BEFORE YOU MOVE ON

Someone reports 45 tokens per second generating with a 13B model at Q4 (about 8 GB) on a laptop with dual-channel DDR5 at roughly 80 GB/s. What should you conclude?

A. It is plausible, since quantised models bypass the memory bottleneck by computing in lower precision.
B. It is plausible if the prompt was long, since long prompts raise generation throughput.
C. The number implies about 360 GB/s of bandwidth, far above the machine's 80 GB/s, so it is probably prompt processing rather than generation.
D. It is plausible because the operating system caches the model, so weights are read from cache rather than memory.

37 · 9 min

Serving many users at once

THE ONE THING TO KEEP

Continuous batching and paged attention let one GPU serve many sequences because weights are read once per step and cache memory is allocated in blocks rather than reserved per request, trading a little per-user latency for several times the total throughput.

Why a different engine exists

Everything so far optimised one conversation. Serving thirty at once is a different problem, and the tool for it is vLLM, or SGLang, or Hugging Face's TGI, or TensorRT-LLM on Nvidia hardware. The ideas below are shared; vLLM is the common starting point.

```
pip install vllm
vllm serve Qwen/Qwen3-8B   --max-model-len 8192   --gpu-memory-
utilization 0.90   --max-num-seqs 64   --enable-prefix-caching
```

Same OpenAI-compatible endpoint as everything else, so your client code does not change.

Continuous batching

A naive server batches requests together, runs them to completion, and starts the next batch. Requests finish at different times, so the batch runs at the speed of its slowest member and the freed capacity sits idle.

Continuous batching schedules at the level of a single decoding step. When one sequence finishes, a waiting request joins the batch immediately. Nothing waits for a batch boundary. Combined with the bandwidth argument from the previous lesson — the weights are read once per step regardless of how many sequences are in the batch — this is where the throughput comes from.

The numbers are dramatic. A single stream on a given GPU might generate 100 tokens per second; the same GPU serving 32 concurrent sequences might produce 2,000 tokens per second in aggregate, with each individual user seeing perhaps 60. Total work goes up manyfold; each person's experience degrades a little.

The same GPU, two scheduling decisions

One stream	Thirty-two streams, continuous batching
Weights read once per token, for one sequence	Weights read once per step, for all 32
About 100 tokens per second in total	About 2,000 tokens per second in total
That one user gets all 100	Each user sees perhaps 60
Cache reserved for the maximum context	Paged attention hands out blocks as needed
Capacity idle between requests	A finished sequence is replaced immediately

Total work rises many times over and each person's experience degrades a little. That is the whole economic basis of hosted inference, and the reason llama.cpp, which optimises one conversation beautifully, is the wrong engine for thirty.

Paged attention

The KV cache was the memory problem in an earlier lesson. A naive implementation reserves the maximum context length for every request, so a request that turns out to be 300 tokens long occupies space for 8,192.

Paged attention allocates the cache in fixed-size blocks, held in a table, exactly as an operating system pages memory. A sequence uses the blocks it needs and returns them when it finishes. Fragmentation and over-reservation mostly disappear, and the number of concurrent sequences the same card supports rises several times over. This is the single largest reason vLLM outperforms simpler servers.

It also makes two features cheap. **Prefix caching** shares the blocks of a common prompt prefix across requests: if every request begins with the same 2,000-token system prompt, it is computed once. On a chat product this is often the biggest single saving available. And **copy-on-write** sharing lets several samples from the same prompt share the prompt's blocks.

The dials

`--max-model-len` is the context limit, and it is a capacity decision as much as a capability one. Serving a 128k model at 8k lets you hold many more users.

`--gpu-memory-utilization` is the fraction of the card vLLM claims, default 0.9. Raise it if nothing else runs on the card; lower it if something does.

`--max-num-seqs` caps concurrent sequences. Higher gives more throughput and worse tail latency.

`--tensor-parallel-size N` splits the model across N GPUs, for models too large for one card. It costs communication between cards on every layer, so use it because you must, not to speed up a model that already fits.

Chunked prefill interleaves long prompt processing with ongoing decoding, so one user pasting a huge document does not stall everybody else's generation. On mixed traffic it is the difference between a smooth service and periodic freezes.

Throughput and latency are the same dial

You cannot maximise both. A larger batch means more total tokens per second and a longer wait for each individual token. Decide which one your product is:

- **Interactive chat.** Time to first token is what people feel. Keep batches moderate, enable prefix caching, and consider streaming so the wait is hidden.
- **Batch processing.** Nobody is watching. Push concurrency until memory limits it, and take the throughput.
- **Mixed.** Run two deployments rather than one compromise, if you can afford the memory. Two servers with different settings usually beat one with settings that suit neither.

When not to reach for this

vLLM wants a GPU, expects Linux, and its advantages appear under concurrency. For one person on a laptop it is slower to set up and no faster in use than llama.cpp, and on CPU-only hardware llama.cpp is the better engine. Choose by your concurrency, not by which project is more discussed.

BEFORE YOU MOVE ON

A chat product with a 2,000-token system prompt on every request moves from a naive server to vLLM with prefix caching and sees a large drop in cost per request. Which mechanism explains most of it?

A. Quantisation applied automatically by vLLM, which reduces the memory read per token.

B. Tensor parallelism, which spreads the shared prompt across GPUs.

C. Continuous batching alone, since more requests fit in each batch.

D. The shared prefix is computed once and its cache blocks are reused across requests, so the repeated prefill work disappears.

38 · 8 min

Running all of this without money

THE ONE THING TO KEEP

Kaggle's weekly GPU hours, Colab's T4, free CPU instances and provider credits are enough to evaluate, fine-tune and batch-process without spending anything, provided every job checkpoints to durable storage and writes results incrementally.

What is genuinely free, and what each is good for

Nothing in this course requires you to buy hardware. Here is the honest state of the free tiers, with their real limits rather than their advertised ones.

Kaggle Notebooks is the most generous and least known. Around 30 GPU-hours per week, on a P100 or a pair of T4s, with sessions up to about 12 hours, plus a large CPU allowance. That is enough to fine-tune a 7B model with QLoRA, run a full evaluation, or process a substantial batch job. If you take one thing from this lesson, it is that this exists.

Google Colab free tier gives you a T4 with 16 GB of VRAM, sessions that disconnect after a few hours and sooner if idle, and no guarantee of a GPU at busy times. Good for a notebook you can restart, poor for anything that must complete.

Hugging Face Spaces hosts free CPU Spaces indefinitely. More usefully, you can *use* other people's public Spaces at no cost, which is how to try six image or speech models in twenty minutes without installing anything.

Oracle Cloud's always-free tier provides ARM instances with a meaningful amount of RAM and no time limit. There is no GPU, so this is CPU inference, but a 7B or 8B model at Q4 runs on it at a few tokens per second, permanently, for nothing. For a background job or a low-traffic personal service that is a real deployment.

Provider free credit. Several inference providers give a small amount of free usage, and Hugging Face includes a monthly credit for its inference partners. Enough to run an evaluation across several models; not enough to serve anything.

Your own machine. An ordinary laptop runs a 3B model at readable speed on CPU alone. This is not a consolation prize; most of the extraction, classification and rewriting work in this course runs there.

What fits where

On a T4 with 16 GB of VRAM:

- Inference on a 7B or 8B model at 4-bit, with room for a long context.
- Inference on a 13B at 4-bit, tight.
- QLoRA fine-tuning of a 7B with a modest sequence length and gradient checkpointing.
- Not: full-precision fine-tuning of anything interesting, or a 70B in any form.

On CPU with 8 to 16 GB of RAM: 3B to 8B at Q4, at the speeds the bandwidth arithmetic predicts.

Working within session limits

The constraint that actually hurts is not memory, it is that free sessions end. Three habits make it survivable, and they are the same habits that make any long job survivable.

Checkpoint to durable storage every N steps. Not to the session's local disk, which disappears. Push adapters to a private repository, or write to mounted cloud storage.

```
from transformers import TrainingArguments
args = TrainingArguments(
    output_dir="out", save_steps=100, save_total_limit=2,
    push_to_hub=True, hub_model_id="you/my-adapter", hub_pri-
vate_repo=True,
)
```

Make the job resumable. A training run that can start from the last checkpoint turns a disconnection into a delay. A run that cannot turns it into a loss.

Write results as you go. One JSONL line per completed item, appended immediately. If the session dies at item 380 of 500, you have 380 results, not none.

These are the same disciplines that matter on paid infrastructure; the free tier just teaches them faster.

What the free tiers are not for

Serving a product. Rate limits, unpredictable availability and terms that prohibit it make this a bad idea, and it will fail at the moment anyone actually uses your thing. For serving, either rent per token, which starts at a few cents, or use the always-free CPU instance and accept its speed.

> *The order that wastes least: evaluate on free provider credit, fine-tune on Kaggle's weekly hours, and only then decide what to pay for. Most people buy hardware first and discover afterwards that a 3B model on their existing laptop did the job.*

BEFORE YOU MOVE ON

A QLoRA fine-tune on a free notebook GPU is disconnected at around hour four of an expected six. What determines whether the work is lost?

A. Whether the model was small enough to fit in VRAM, since larger models fail to save state.
B. Whether checkpoints were written to durable storage outside the session and the run can resume from the last one.
C. Whether the session was on Colab or Kaggle, since one preserves state across disconnects.
D. Whether gradient checkpointing was enabled, which stores intermediate states for recovery.

39 · 9 min

Rent, or own: the arithmetic

THE ONE THING TO KEEP

Breakeven against a rented GPU sits in the hundreds of millions of tokens a month and assumes high utilisation, so the real reasons to self-host are usually privacy, offline operation and permanence rather than cost.

Three ways to buy inference

Per token, from an inference provider. You pay for what you use, there is nothing to operate, and the price already includes their batching efficiency.

Per hour, renting a GPU from a cloud. You pay whether or not anything is running.

Outright, buying hardware. You pay once, plus electricity, plus your time.

The decision is arithmetic plus three things arithmetic cannot capture. Do the arithmetic first, because it eliminates most cases.

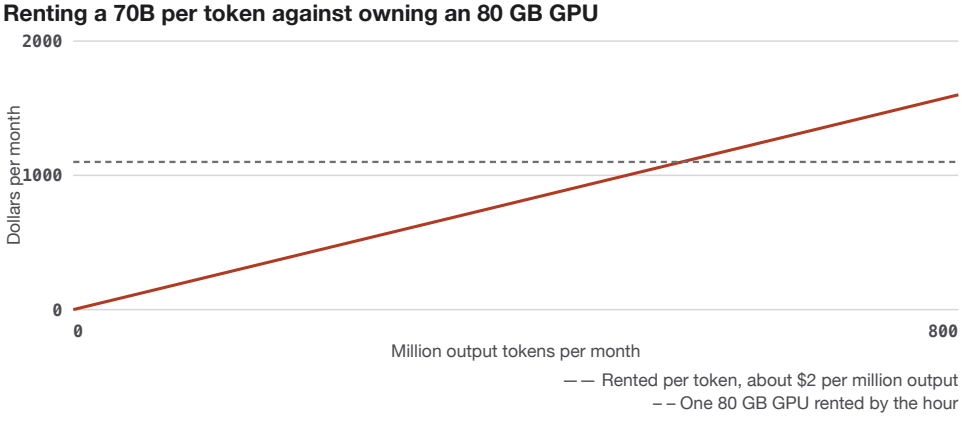
The numbers, at the time of writing

Per-token prices for open models sit roughly at a few tens of cents per million tokens for an 8B class model and a few dollars per million for a 70B, with output priced two to four times input, and a three to five times spread between providers for the identical model.

An A100 with 80 GB rents for somewhere between one and two dollars an hour depending on the provider and commitment, which is roughly 700 to 1,500 dollars a month running continuously. A consumer card such as a used 3090 with 24 GB costs a few hundred to about a thousand to buy, and draws 300 to 450 watts under load — at typical electricity prices, ten to thirty dollars a month if it runs all day.

The breakeven, and why it usually favours renting

Take a 70B model at roughly two dollars per million output tokens rented per token, against an 80 GB GPU at about 1,100 dollars a month. Breakeven is around 550 million output tokens a month.



Breakeven is around 550 million output tokens a month, roughly 18 million a day, every day. Most products are two orders of magnitude below it. And the flat line assumes the card is busy: at five per cent utilisation it costs twenty times its sticker price per token delivered.

That is a very large number. It is roughly 18 million tokens a day, or about 7,000 substantial responses an hour, every hour, all month. Most products are two orders of magnitude below it.

And that comparison assumes you keep the GPU **busy**, which is the assumption that fails. A GPU at 5% utilisation costs twenty times its sticker price per token delivered. Real traffic is bursty: quiet at night, peaked in the afternoon. A rented endpoint absorbs that for free because someone else's traffic fills the gaps. Your own machine does not.

So the honest default is: **rent per token until you have steady, high, predictable load.** The exceptions are not about cost.

The three reasons that are not arithmetic

Data cannot leave. Health records, legal files, recordings of conversations, anything under a data residency obligation or a contract you signed. This is the most common genuine reason to self-host and it does not need a cost justification, though it is worth stating the cost honestly so the decision is made with open eyes.

It must work offline. A factory floor, a field device, a clinic with intermittent connectivity, a laptop on a plane. No amount of price advantage helps a service that is unreachable.

Nobody can take it away. A rented model can be deprecated, reconfigured, price-changed or restricted. Weights on your disk cannot. For a product whose behaviour must be stable for years, this is a real form of insurance.

There is a fourth that people undercount: **learning**. Running the stack yourself teaches you what the previous six lessons describe. That has value even when it is not the cheapest option.

The costs people leave out

When someone presents a spreadsheet showing self-hosting is cheaper, check for these:

- **Utilisation.** Divide by the fraction of the month the machine is actually working. This is usually the whole error.
- **Redundancy.** One GPU is a single point of failure. Two doubles the cost.

- **The person.** Somebody has to patch, monitor, respond at 3am and handle upgrades. A fraction of an engineer costs more than the hardware.
- **Model refresh.** A better model arrives every few months. Re-quantising, re-evaluating and re-deploying is recurring work.
- **Idle time in development.** The machine you bought for production is also the one people leave a notebook running on.

A decision rule you can use today

1. If data may not leave, self-host and stop reading.
2. If it must work offline, self-host and stop reading.
3. Otherwise, estimate monthly tokens honestly, multiply by a provider's price, and compare against a rented GPU at your *realistic* utilisation, not at 100%.
4. If renting per token is cheaper, rent, and revisit at ten times the volume.
5. Buy hardware for learning, for a workstation, or for genuinely constant load — not to save money on a service that runs for two hours a day.

BEFORE YOU MOVE ON

A team argues that renting a GPU at 1,100 dollars a month beats paying 900 dollars a month to a per-token provider. Their traffic runs about six hours a day on weekdays. What is wrong with the comparison?

A. Nothing; the fixed cost is higher but the GPU also removes network latency, which is worth the difference.

B. The GPU is idle around 82% of the time, so its effective cost per delivered token is several times what the monthly figure suggests, while the provider bills only for use.

C. The provider's price will fall over time, so the comparison should use projected rather than current prices.

D. A rented GPU cannot serve the same model quality as a provider, since providers use larger hardware.

40 · 8 min

What actually runs on a phone

THE ONE THING TO KEEP

Mobile operating systems cap per-application memory, which limits on-device models to roughly 1 to 3 GB, and sustained generation is halved by thermal throttling — so a phone model is a private text-transformation engine, not a knowledgeable assistant.

The constraint is not the chip

A recent phone has a capable processor and 8 to 16 GB of RAM. What it does not have is permission to use it. Mobile operating systems cap how much memory a single application may hold — on iOS the ordinary limit is a fraction of physical RAM, raised only with a specific entitlement — and a process that exceeds it is terminated without ceremony.

So the practical budget for a model on a phone is roughly **1 to 3 GB**, which means a 1B to 3B model at 4-bit. Everything else about on-device inference follows from that number.

What that size can do, on a phone

Realistically good: rewriting a message, summarising a notification or a short email, classifying and tagging, extracting a date and an amount from a text, offline translation of short passages with a language-tuned model, autocomplete, and simple structured extraction.

Realistically bad: anything needing world knowledge, long documents, code, or multi-step reasoning. A 3B model on a phone is a text-transformation engine, not an assistant that knows things.

That is not a small category. Most of the useful private computing on a phone is text transformation, and doing it on the device rather than on a server is ex-

actly the case where privacy is the point.

The routes

llama.cpp under Termux on Android is the direct path: install Termux, install the package, run `llama-cli` with a GGUF from your download folder. It works, it is fiddly, and it is the fastest way to see real numbers on your own hardware.

MLC LLM compiles models for mobile GPUs and ships a chat application for both platforms.

Google AI Edge and the MediaPipe LLM inference API give an Android-native path with models packaged for it.

Apple's on-device foundation model, exposed to applications through a framework, is the sanctioned route on iOS and avoids the memory limit entirely because the model belongs to the system rather than your app. You do not choose the model; that is the trade.

ExecuTorch is the PyTorch path to a compiled on-device artefact, and is the most work.

Prebuilt applications — PocketPal, ChatterUI and others — let you load a GGUF from the phone's storage and chat with it. This is the right first step: measure before you build.

Thermal throttling, which nobody mentions

A phone has no fan. Sustained generation heats the chip, the operating system reduces clock speeds, and speed falls. The characteristic pattern is 15 to 20 tokens per second for the first thirty seconds, settling to half that after a minute or two of continuous work, on a device that is also getting uncomfortably warm.

Design around it. Short bursts are fine. A background job that summarises fifty documents is not a phone task, or if it is, it belongs on the charger overnight with expectations set accordingly. Battery drain is proportionate: continuous generation is comparable to gaming.

The NPU question

Phones have neural accelerators, and the marketing numbers for them are large. Most general-purpose local inference does not use them. Reaching an NPU means going through a vendor-specific stack — Qualcomm's on Android, Apple's Neural Engine through Core ML — with model conversion, operator support limits and fixed shapes. Portable engines such as llama.cpp use the CPU and sometimes the GPU instead.

This is improving, but if you read a claim of very high on-device throughput, check whether it required a vendor toolchain and a specific converted model. It usually did.

Deciding whether to bother

Ask what the device path buys you.

- **Privacy is the requirement** — messages, health notes, photographs. Strong reason; the data never leaves.
- **It must work with no network** — travel, field work, poor coverage. Strong reason.
- **Latency for a tiny task** — a 200-millisecond rewrite with no round trip feels different from a 900-millisecond one. Real, and often underrated.
- **Cost** — weak. A 3B model called a few times per user per day is very cheap to serve centrally, and you avoid shipping a 2 GB download in your application.

The honest summary: on-device inference is a genuine capability with a narrow, well-defined scope. Within that scope it is excellent, and it is the only place where you can promise a user that their words never left the phone and mean it.

BEFORE YOU MOVE ON

A benchmark shows a phone generating 18 tokens per second with a 3B model, but users report it feeling much slower in a long session. What is the most likely explanation?

A. Users are running other applications, so the model is being paged out of memory between requests.

B. The benchmark measured prompt processing, which is much faster than generation.

C. The phone has no active cooling, so sustained generation heats the chip, the system reduces clock speed, and throughput falls to roughly half after a minute.

D. Battery saver mode reduces the model's precision, degrading both speed and quality.

CASE STUDY · MODULE 4

Eleven branches, one radiologist, and data that cannot leave

A diagnostic-imaging chain in Nagpur with eleven branches, trying to cut the two-day delay between a radiologist dictating a report and a patient collecting it.

The bottleneck at Shreeya Diagnostics was not the scan. It was the typing. Radiologists dictated findings into a handset; three typists across two shifts transcribed them; a radiologist read the typed version, corrected it and signed. The median delay from dictation to signature was thirty-one hours, and almost all of it was queue.

The proposal from their software vendor was to transcribe the dictation automatically and lay it out in the house report format, so the radiologist received a draft to correct and sign rather than a blank page. Nobody involved suggested the model should produce a finding or that anything would reach a patient unsigned. The whole value was in removing typing from the critical path.

The arithmetic that did not decide it

The technical director, Anjali, did the rented-per-token arithmetic first, because it is cheap and it eliminates most cases. Across eleven branches they generated about nine thousand minutes of dictation a month. Transcribed and laid out, that came to roughly twelve million tokens of input and four million of output — a few hundred rupees a month at aggregator prices, against a 24 GB card that would cost eighty thousand rupees and draw power in a room already fighting the heat.

Renting was not close. It was something like eight times cheaper and it required nothing to operate.

It was also not available. Their contract with the tertiary hospital whose outpatients they scanned named the data as confidential and restricted where it could be processed, and a recording of a radiologist describing a named patient's chest is about as identifiable as clinical data gets. Anjali wrote the arith-

metic into the decision memo anyway, with the sentence her chief executive later quoted back at her: we are choosing to spend eight times more, and here is exactly how much more, so that nobody in two years thinks this was free.

The decision inside self-hosting

That settled where it ran and opened the real question: one machine or eleven.

One server at head office. A 24 GB card running speech recognition with voice-activity detection in front of it, and an 8B model at four-bit for the layout and formatting pass, served with continuous batching so eleven branches share it. Anjali's arithmetic said the card would sit at perhaps six per cent utilisation, which is a poor use of eighty thousand rupees and completely irrelevant, because the alternative was not a cheaper card. The problem was the link: four of the eleven branches were on connections that dropped for hours in the monsoon, and a branch that cannot reach head office falls back to typing, which is the situation they were trying to leave.

A small box per branch. No GPU. A 3B model at four-bit and a fast reimplementation of the speech model at eight-bit integer, both on the CPU. Anjali predicted the speed before buying anything: the branch machines had about fifty gigabytes a second of memory bandwidth and the model file was two gigabytes, so the ceiling was twenty-five tokens a second and the realistic figure fifteen to eighteen. A four-hundred-token report draft would take under half a minute. That is not fast and it is very far inside thirty-one hours.

Eleven boxes cost less in total than one good card. They also meant eleven machines to patch, eleven model files to keep at the same revision, and eleven places for someone to notice that the quality had changed.

What made the smaller model defensible

The 3B was worse than the 8B at the layout task and Anjali could say by how much: on forty real dictations, ninety-one per cent of drafts needed no structural correction against ninety-six per cent for the 8B. Both numbers were inside the noise of a forty-item set and she said so in the memo. What mattered

more was the failure mode. The smaller model's errors were clerical — a heading in the wrong order, a measurement left in the wrong section — and visible to the radiologist reading the draft. Neither model was asked to decide anything, and both drafts went to a person who signed.

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 put a box in every branch. Each ran voice-activity detection ahead of the transcription model, which mattered more than any other single choice: dictation is full of pauses, and without the detector the transcriber invented sentences during silence — polite, fluent, entirely fabricated lines that a tired radiologist might have signed. With it, silence produced nothing. The median dictation-to-signature time fell from thirty-one hours to under four, and the typists moved to checking and to the front desk rather than leaving. Two things went in the runbook: the model files are pinned by hash and updated from one central copy so all eleven branches run the same revision, and the eight-times-cheaper rented arithmetic stays in the memo, revisited annually, so the decision remains a decision rather than an inheritance.

Renting was roughly eight times cheaper and was rejected. Why did Anjali put the arithmetic in the memo anyway?

Because a constraint that is never priced turns into folklore. Writing down that self-hosting costs eight times more keeps the decision reviewable: if the contract changes, or a provider offers processing that satisfies the residency obligation, the comparison is already made and someone can revisit it in an hour. It also protects the decision in the other direction — nobody can later argue that self-hosting was chosen to save money, which would be false and would justify the wrong things.

She predicted fifteen to eighteen tokens per second for the branch machines before buying one. How, and why is that prediction more

useful than a benchmark chart?

Decoding reads every active weight from memory once per token, so tokens per second is bounded by memory bandwidth divided by the model file size: about fifty gigabytes a second divided by a two-gigabyte file gives a ceiling of twenty-five, and real throughput is typically sixty to eighty per cent of the ceiling. It is more useful than a chart because it applies to the exact machine and file she was considering, costs nothing, and can be done before any download — and because it tells her which lever to pull if the answer is too slow, namely a smaller file rather than a faster processor.

Voice-activity detection turned out to be the most important component in the pipeline. Why, and what does that suggest about evaluating a transcription system on clean samples?

Because the transcription model produces its most likely continuation whatever the audio contains, and on silence or background noise that continuation is often a fluent sentence drawn from common subtitle text. Dictation is mostly pauses, so the failure is frequent and invisible: the transcript reads well and contains sentences nobody said. Evaluating on clean, continuous samples hides it entirely. The evaluation has to include the recordings you actually get, with their pauses, their room noise and their interruptions.

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 An eight-billion model at Q4 is about 4.9 gigabytes rather than 4.0. Where does the extra gigabyte come from?

A. The tokenizer and configuration files packed inside the single GGUF file

B. The per-block scale factors stored alongside the four-bit indices

C. The KV cache, which is allocated inside the model file at load time

D. Padding added so that every tensor aligns to the memory page size on disk

2 You have about eight gigabytes for weights. A thirteen-billion model at Q4 and a seven-billion model at Q8 both fit. Which is usually better?

A. The thirteen-billion at Q4, since larger models carry redundancy that absorbs the error

B. Neither; at the same file size two models always perform within noise of each other

C. The seven-billion at Q8, because precision matters more than parameters below sixteen bits

D. The seven-billion at Q8, because a model below eight bits loses its instruction tuning

3 Someone reports sixty tokens per second from a five-gigabyte model on a laptop with fifty gigabytes per second of memory bandwidth. Why is that implausible?

A. Quantised models run slower than full precision because of the unpacking step

B. Tokens per second is limited by arithmetic throughput, which a laptop does not have

C. Laptops throttle after about thirty seconds, halving any sustained rate

D. Decoding reads every active weight once per token, capping it near ten per second

4 A benchmark reports 900 tokens per second for prompt processing and 12 for generation, on the same machine and the same file. What explains the gap?

A. Prompt tokens are shorter than generated tokens, so more of them fit per second

B. Prompt processing is measured before the model is fully loaded into memory

C. Prefill reads the weights once per prompt; decode reads them per token

D. Generation is slowed by the sampler, which scores the whole vocabulary each step

5 A colleague says a local model answers well about the end of a long document and seems not to have read the beginning. What do you check before blaming the model?

A. Whether the document was pasted in with the wrong character encoding

B. The context length the wrapper set, which has often defaulted to 2048

C. Whether an instruct model was loaded rather than a base model

D. Whether the quantisation damaged long-context retrieval at this bit width

6 A spreadsheet shows self-hosting a seventy-billion model is cheaper than renting per token. Which omission most often reverses the conclusion?

A. Utilisation: the card is paid for all month and busy for part of it

B. The download cost of the weights, which recurs with every model refresh

C. Output tokens being priced two to four times input tokens on rented endpoints

D. Electricity, which at datacentre rates exceeds the rental price of the card

MODULE 5

Getting good output

An open model arrives with none of the scaffolding a commercial product puts around it: no hidden system prompt, no server-side reformatting, no safety layer, no tool harness. That is the freedom and it is also the work. This block covers the sampler, constrained decoding, prompting that survives a small model, examples, tool calls, agent loops, other languages, refusals, guardrails, and what to do when the model does not know.

> BY THE END OF THIS MODULE YOU CAN
>
> Take an open model producing bad output and identify whether the fault lies in the sampler, the prompt, the examples, the schema, the language or the model itself — then apply the cheapest correct fix, from a temperature change to a grammar-constrained decoder to a separate guardrail model, and demonstrate the improvement on a held set

41 · 9 min

The sampler, and the defaults that ship broken

THE ONE THING TO KEEP

The sampler converts logits into a token, and the common default of a repetition penalty above 1 actively damages JSON and code because those formats require repeated punctuation tokens.

What happens after the forward pass

The model outputs a logit for every token in the vocabulary — 150,000 raw scores. The sampler turns those into one chosen token, and every parameter you have heard of is a step in that conversion.

Temperature divides the logits before the softmax. Below 1 the distribution sharpens and the likeliest tokens dominate; above 1 it flattens and unlikely tokens get a chance. At 0 most implementations skip sampling entirely and take the highest-scoring token, which is greedy decoding.

top_k keeps only the k highest-scoring tokens and renormalises. Blunt: k=40 is far too many when the model is certain and far too few when it is genuinely uncertain across a wide field.

top_p, or nucleus sampling, keeps the smallest set of tokens whose probabilities sum to p. Adaptive in the right direction — a confident step keeps two tokens, an open one keeps forty.

min_p keeps tokens whose probability is at least some fraction of the top token's probability. At min_p 0.05, if the best token has probability 0.6, nothing below 0.03 survives. It behaves better than top_p at high temperature, which is why creative-writing setups have converged on it.

repetition_penalty divides the logits of tokens that have already appeared, with values above 1 discouraging them. **frequency_penalty** and **presence_penalty** do a similar job additively.

The default that quietly ruins structured output

Most local interfaces ship with something like temperature 0.8, top_p 0.95 and a repetition penalty around 1.1. That is a reasonable configuration for chat and a bad one for everything else, and the repetition penalty is the actively harmful part.

Consider what it does to JSON. The tokens `"`, `:`, `,`, `{` and `}` must repeat — that is what the format is. A repetition penalty pushes their logits down every time they appear, so by the fifth field the model is being actively discouraged from producing correct syntax. The same applies to code, where indentation and brackets repeat by necessity, and to any tabular or list output.

Set repetition penalty to 1.0 for structured output. If the model genuinely loops, the cause is usually the prompt or a missing stop token, and the penalty is treating the symptom by damaging the format.

Two sets of sampler settings, and why the shipped defaults suit neither

Extraction, classification, routing, schemas	Writing, brainstorming, variation
temperature 0	temperature 0.7 to 1.0
top_p 1, top_k 0	min_p about 0.05, or top_p 0.9
repetition_penalty 1.0	repetition_penalty 1.05 to 1.1 is defensible
The same input must give the same output	Reasoning models: follow the card, often 0.6

Most local interfaces start at temperature 0.8, top_p 0.95 and a repetition penalty near 1.1. Quotes, colons, commas and braces are what JSON is made of, so that penalty is arguing against correct syntax by the fifth field.

Settings that work

Extraction, classification, routing, schema-filling. Temperature 0, top_p 1, top_k 0, repetition penalty 1.0. You want the same input to give the same output, and you want the model's best guess rather than a sample from its uncertainty.

Factual question answering over provided context. Temperature 0 to 0.2. Same reasoning.

Writing, brainstorming, variation. Temperature 0.7 to 1.0 with min_p around 0.05, or top_p 0.9. A repetition penalty of 1.05 to 1.1 is defensible here, where the format is prose.

Reasoning models. Follow the model card, and note that several of them explicitly recommend *against* temperature 0 — often something like temperature 0.6 with top_p 0.95. Greedy decoding on a long chain of thought is prone to getting stuck in loops, because one bad deterministic step repeats forever. This surprises people who have learned that zero is the safe default.

Determinism, honestly

Temperature 0 gives you the same token sequence for the same input on the same machine with the same batch composition. It does not guarantee bit-identical results across different hardware, different batch sizes, or different library versions, because floating-point addition is not associative and kernels reduce in different orders.

For evaluation this rarely matters, since the differences are tiny and only occasionally flip a near-tie. For a regression test that asserts exact strings it matters a great deal, and the fix is to assert on the parsed result rather than on the raw text.

The rule to carry away

Record the sampler with every result. Two comparisons of the same models can disagree entirely because one was run at temperature 0.8 and the other at 0. It is three numbers in a log line, and it turns an argument into a fact.

```
{"model":"qwen3-8b","rev":"9a3f21c","quant":"Q4_K_M",

"temperature":0,"top_p":1,"top_k":0,"repeat_penalty":1.0,"max_-
tokens":512}
```

BEFORE YOU MOVE ON

A model producing JSON starts emitting malformed output after the fourth field — missing quotes, dropped commas — but produces valid JSON for short objects. What should you suspect first?

A. A repetition penalty above 1.0 is progressively suppressing the punctuation tokens the format requires.

B. The context window is being exceeded, truncating the later fields.

C. Temperature is too high, so the model is choosing unlikely tokens.

D. The model is too small to hold a schema in mind for more than a few fields.

42 · 9 min

Making the output valid by construction

THE ONE THING TO KEEP

Grammar-constrained decoding masks illegal tokens at each step so invalid syntax becomes impossible, which converts parse failures into silently wrong values unless the schema gives the model an explicit way to express absence.

Asking is not enough

Ask a 7B model to reply with JSON and it will, most of the time. The rest of the time it adds a sentence before the brace, wraps the object in a code fence, uses single quotes, or trails off. Retry loops and regular expressions to pull the object out are the usual response, and they are a bad answer to a problem with a good one.

Constrained decoding removes the failure entirely. At every generation step, a state machine tracks where you are in the grammar, works out which tokens could legally come next, and sets the logits of every other token to negative infinity before sampling. An invalid character is not unlikely; it is impossible.

Using it

llama.cpp, directly from a JSON Schema:

```
curl localhost:8080/v1/chat/completions -d '{
  "messages":[{"role":"user","content":"Extract from: Invoice
8841, 12 March, EUR 4,200, Meridian Ltd"}],
  "temperature":0,
  "response_format":{"type":"json_schema","json_schema":
{"schema":{
    "type":"object",
    "properties":{
      "invoice_no":{"type":"string"},
      "date":{"type":"string","format":"date"},
      "currency":{"type":"string","enum":
["EUR","USD","GBP","INR"]},
      "amount":{"type":"number"},
      "supplier":{"type":"string"}},
    "required":["invoice_no","currency","amount"],
    "additionalProperties":false}}}}'
```

vLLM takes `guided_json`, `guided_choice` and `guided_regex`. **Outlines**, **xgrammar** and **lm-format-enforcer** are the libraries underneath most of these, and can be used directly with `transformers`.

llama.cpp also accepts a raw grammar in its own GBNF notation, which is the escape hatch when a schema cannot express what you need:

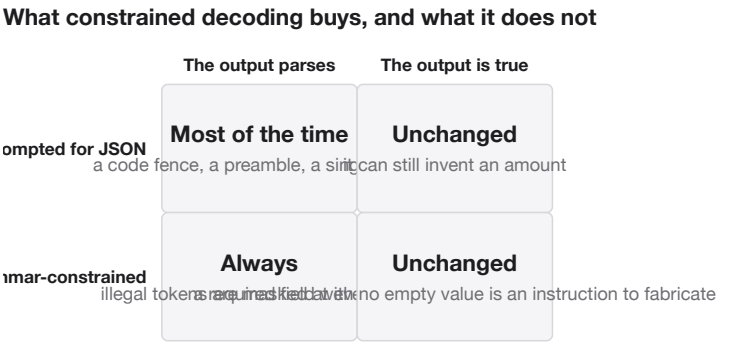
```
root   ::= "SENTIMENT: " label "\nREASON: " [^\n]+ "\n"
label  ::= "positive" | "negative" | "neutral"
```

What it guarantees and what it does not

It guarantees **syntax**. The output will parse, the enum will be one of your values, the number will be a number.

It guarantees nothing about **truth**. A model forced to produce an amount will produce an amount, whether or not the document contains one. Constrained decoding converts a parse failure into a confidently wrong value, which is

harder to notice. That trade is usually worth making, and you have to know you made it.



Constraining converts a parse failure into a confidently wrong value, which is harder to notice. The trade is usually worth making and you have to know you made it: make the field nullable, or add an explicit unknown to the enum, so the model has somewhere to go.

Three consequences follow.

Always give the model a way to say nothing. If a field may be absent, make it nullable or add an explicit `"unknown"` to the enum. A required field with no legal empty value is an instruction to fabricate.

Order the fields deliberately. Generation is left to right, so a field produced early conditions everything after it. Putting a short `evidence` or `reasoning` string before the answer field lets the model work before it commits; putting it after means the reasoning is a justification of a decision already made. This is a real and easily measured effect.

Keep schemas shallow. Deeply nested objects with many optional branches are harder for small models, and the grammar machinery gets slower to compile. Flat objects with clear names work far better at 7B than an elegant nested design.

Costs

Compiling a grammar takes time on first use, so cache it rather than rebuilding per request. Per-token overhead is small in modern implementations. The larger cost is subtler: a heavily constrained format can pull the model off the

distribution it writes well in, so quality of the *content* can drop slightly even as validity goes to 100%. Measure both.

When not to constrain

Free-form prose, and any case where you would rather see the model fail visibly than produce a well-formed wrong answer. For a first exploration of a new task, run unconstrained and read what the model naturally produces; it tells you what it understood. Constrain afterwards, once you know the shape you want.

> *The practical upgrade path: prompt for JSON, discover the failure rate on 200 real inputs, then constrain and watch it go to zero. Skipping the measurement means you never find out whether the schema you wrote matches the task, because the errors that remain are now silent.*

BEFORE YOU MOVE ON

After enabling JSON Schema constrained decoding, an extraction pipeline's parse errors drop to zero but downstream data quality gets worse. What is the most likely mechanism?

A. Constrained decoding lowers the model's effective temperature, reducing the diversity needed for accurate extraction.

B. Required fields with no null or unknown option force the model to emit a value even when the document contains none, so failures become plausible wrong answers instead of errors.

C. The grammar compilation adds latency, causing timeouts that truncate results.

D. JSON output uses more tokens than free text, so the model runs out of context.

43 · 9 min

What changes when nothing is hidden

THE ONE THING TO KEEP

With open weights you supply the scaffolding a hosted product hides, so instructions belong at the end of the prompt, phrased positively, one job per call — and prompts must be versioned per model because they do not transfer between families.

You are now the scaffolding

A commercial chat product is not just a model. It is a model plus a long system prompt you never see, plus server-side reformatting, plus a tool harness, plus a safety layer, plus retries. When you download weights, you get the model and none of the rest.

That is the trade this whole course is about, and in prompting it shows up as a set of specific differences.

No hidden system prompt. Whatever behaviour the product had — cite sources, ask clarifying questions, refuse this category, format with headings — was partly instruction, not weights. You have to write it.

No reformatting. The exact string you send is what the model sees, special tokens and all. There is no service quietly fixing your template.

Weaker system-prompt adherence at small sizes. A 4B model follows a system prompt for a while and then drifts back to its tuned defaults. This is not a bug you can prompt away; it is what a smaller instruction-tuned model does.

Six things that measurably help

Put the critical constraint last. Attention to recent tokens is stronger, and small models in particular follow the end of the prompt more reliably than the

middle. If one rule must hold, restate it immediately before the model generates.

Write positively, not negatively. *Do not mention the price* frequently produces a mention of the price, because the phrase puts the concept in context and small models are weak at the logical operation of suppression. *Discuss only availability and delivery* works better. This is one of the most reliable differences between prompts that work at 4B and prompts that do not.

Be literal about the output. Show the exact shape you want, including an example. Small models copy structure well and infer it badly.

One job per call. A prompt that classifies, extracts and summarises will do one of the three well. Three calls to a 4B model are usually cheaper and better than one call to a 30B, and each is separately testable.

Give a decision procedure, not a description. *If the message names a product and a defect, label it fault; if it names a payment or a refund, label it billing; if both, prefer billing*. Ordered rules beat adjectives.

Repeat the format at the end. A single line — `Reply with only the JSON object.` — placed after the input rather than before it, is worth more than three sentences of instruction at the top.

Prompts do not transfer

A prompt tuned against one family will underperform on another, sometimes badly. Different instruction-tuning data taught different conventions: how strongly a system prompt binds, whether Markdown headings are natural, how the model handles a numbered list of rules, whether it prefers to explain before answering.

This has an organisational consequence people resist. **Keep the prompt per model in your repository**, versioned next to the model revision. When you swap models, re-tune and re-measure. Treating the prompt as portable is how a model swap gets blamed for a regression that was really a prompt mismatch.

```
prompts/
  classify.qwen3-8b.md
  classify.gemma3-4b.md
  classify.llama31-8b.md
```

Where the system prompt goes when there is none

Gemma has no system role. Several other models were tuned with very few system messages and follow them weakly. Read the chat template: if `system` does not appear in the Jinja source, prepend your instructions to the first user message instead, and expect them to bind about as strongly.

Iterate the way you would test code

Fifteen development examples. Change one thing. Run at temperature 0. Count. Keep a log of what you tried and what it scored.

Two failure patterns to watch for. **Prompt bloat**: a prompt that has grown to 800 tokens through accumulated patches, half of which no longer do anything. Delete a paragraph and re-measure; you will often find the score unchanged and the cost lower. And **overfitting to the fifteen**: hold back another twenty-five examples you touch once at the end, or you have measured your own tuning rather than the prompt.

> *The most useful habit: when a prompt fails, print exactly what the model received, including the applied template. About a third of prompting problems turn out to be formatting problems wearing a prompting costume.*

BEFORE YOU MOVE ON

A prompt containing the line 'do not mention competitors' works on a 32B model but produces competitor mentions on a 4B. What is the best fix?

A. Raise the instruction's priority by moving it into the system prompt and marking it as important.
B. Lower the temperature so the model is less likely to stray into forbidden topics.
C. Rewrite it as a positive constraint naming what to discuss, and place it immediately before generation begins.
D. Add a post-processing filter that removes competitor names from the output.

44 · 8 min

Examples, and how to choose them

THE ONE THING TO KEEP

Examples mainly teach the output format and the label space rather than the mapping, so consistency of format and balanced coverage of labels matter more than how subtle the individual answers are.

Why examples help small models more

A large instruction-tuned model has strong priors about what you probably want. A small one does not, so showing it three worked cases moves it further. On classification and extraction tasks, going from zero examples to three or five is often a bigger improvement than doubling the model size, and it costs a few hundred tokens.

What the examples actually convey is worth understanding, because it changes how you choose them. Research on in-context learning found that corrupting the *labels* in the examples — pairing inputs with wrong answers — degraded performance far less than expected, while removing the examples

entirely, or changing their format, hurt a lot. The examples are mostly teaching the model three things: what the input space looks like, what the label space is, and exactly what shape the output should take. The correct mapping is carried less than intuition suggests.

That does not mean you should use wrong labels. It means the highest-return properties of an example set are **format consistency** and **coverage of the label space**, not subtlety of the individual answers.

How many

Three to eight for most tasks. The curve flattens quickly, and every example is context you pay for on every request. Twenty examples in the prompt is usually a sign that the task should be fine-tuned instead — a later block covers exactly that trade.

Six rules for choosing them

Cover every label at least once. A model shown four billing examples and one fault example will over-predict billing. This majority-label bias is strong at small sizes and is the most common defect in hand-written example sets.

Include the hard cases, not the clear ones. Examples that a rule could have handled teach nothing. The example worth its tokens is the ambiguous one where you show which way to resolve it.

Keep the format byte-identical across examples. Same field order, same separators, same capitalisation. Any inconsistency is a licence for the model to be inconsistent too.

Watch the ordering. The last example has the most influence. Do not put all of one label at the end, and if you rotate examples, keep the order fixed within a run so results are comparable.

Use real inputs. Invented examples are cleaner, better punctuated and more polite than what users actually send, and a model tuned on them handles real messiness worse.

Include one example of the boundary case. What to do when the input is empty, off-topic, or unanswerable. Without it the model has never seen a refusal in your format and will not produce one.

Selecting examples per request

For a task with diverse inputs, a fixed set of five examples is a compromise. Retrieving examples similar to the current input usually does better:

```
# index once
emb = model.encode([e["input"] for e in pool], normalize_embed-
dings=True)

# per request: take the 4 nearest labelled examples
sims = emb @ model.encode([user_input],
normalize_embeddings=True).T
picked = [pool[i] for i in sims.ravel().argsort()[-4:]]
```

This is a small retrieval system in front of your prompt and it is often worth several points on a heterogeneous task. Two cautions: keep the label distribution of what you retrieve balanced, or a query that resembles many billing tickets pulls four billing examples and biases the answer; and put the selected examples in a fixed order so runs stay comparable.

Measuring instead of guessing

Example choice is easy to test and rarely tested. On your fifteen development items:

1. Zero examples. Record the score.
2. Three fixed examples covering all labels. Record.
3. Eight fixed examples. Record.
4. Four retrieved examples. Record.

Half an hour, four numbers, and you will usually find that the difference between three and eight is smaller than the difference between a balanced set

and an unbalanced one. That result is more useful than any general advice, including this lesson's.

> *The one-line summary a colleague can act on: examples set the format and the label space, so make them consistent, balanced and real, and spend your attention on coverage rather than on cleverness.*

BEFORE YOU MOVE ON

A five-example prompt for a three-way ticket classifier contains four billing examples and one fault example. The model over-predicts billing. What is the most direct fix?

A. Add a sentence telling the model that the classes are equally likely.

B. Balance the example set so each label appears, since examples convey the label space and an unbalanced set skews the prior.

C. Increase to fifteen examples so the imbalance matters less.

D. Remove the examples entirely, since they are causing the bias.

---

45 · 9 min

Tool calling, and what breaks at 8B

THE ONE THING TO KEEP

Tool calling is a trained output format plus a matching server-side parser, and small models fail specifically on long tool lists, nested arguments and deciding not to call — so validate arguments and return errors as tool results rather than exceptions.

The mechanism

Tool calling is not a special capability of the runtime. It is a format the model was trained to emit and the server was written to parse.

Your tool definitions go into the prompt, usually as JSON inside a system turn shaped by the chat template. The model, when it decides to call one, emits a specific pattern — a `<tool_call>` block for some families, a JSON object with particular keys for others, a Python-like call for others still. The serving layer recognises that pattern, extracts the call, hands it back to your code, and your code runs the function and returns the result as a new message with a `tool` role.

Two things must line up: the model must have been trained for tool use, and your server must implement that model's parser. In vLLM you name it explicitly:

```
vllm serve Qwen/Qwen3-8B --enable-auto-tool-choice --tool-call-
parser hermes
```

This is why the identical client code produces perfect tool calls on one model and raw text containing a JSON blob on another. Nothing is broken; the parser does not match the format.

What works at 8B

- **One to three tools**, with names that describe the action plainly.
- **Flat arguments** — strings, numbers, enums. No nested objects.
- **A single call per turn**, then your code returns the result and the model continues.
- **Clear disjoint purposes.** Two tools that could both plausibly apply produce coin-flip selection.

What fails, reliably

Many tools. Selection accuracy falls off sharply as the list grows; beyond about ten tools a small model is close to guessing between similar ones. The fix is a routing step: use a cheap classifier or a retrieval step to select five candidate tools, then present only those.

Nested arguments. A parameter that is an object with its own fields is where malformed calls appear. Flatten it, even at the cost of an ugly interface.

Parallel calls. Emitting several calls at once is supported by some templates and unreliable at small sizes. Ask for one at a time.

Deciding not to call. A model given a hammer uses it. Questions answerable from general knowledge get a search call anyway. This needs an explicit instruction and an example of a no-call turn.

Argument fidelity. Dates get reformatted, identifiers get normalised, a currency gets dropped. Always validate the arguments against your schema before executing.

The pattern that makes it robust

Validate, and return failures to the model as tool results rather than as exceptions:

```
try:
    args = ToolArgs.model_validate_json(call.arguments)   # py-
dantic
    result = run(args)
except ValidationError as e:
    result = {"error": "invalid arguments", "detail":
e.errors()[:3]}

messages.append({"role":"tool","tool_call_id":call.id,"content"
:json.dumps(result)})
```

A model that receives a clear error usually fixes its call on the second attempt. A model that receives nothing loops. Cap the retries at one or two, then fail loudly.

Two further safeguards worth building in from the start. **Make every tool idempotent or guarded**, because the model will occasionally call the same one twice. And **never give a small model a tool that performs an irreversible action without a confirmation step** — deleting, sending, paying. The compounding-error arithmetic from earlier applies directly, and a

92% correct tool selection is a 8% chance of an unwanted irreversible act per call.

The alternative that always works

If a model has no tool-calling training, or your server has no parser for it, you do not need either. Ask for constrained JSON and dispatch it yourself:

```
{"action": "search_orders", "customer_id": "...", "since":
"2026-01-01"}
```

With a schema-constrained decoder the output is guaranteed valid, you write the dispatch in ten lines, and it works identically on every model including base models. You lose the multi-turn conventions the trained format provides, and for a one-shot decision that is no loss at all.

> *Start there when you are evaluating models. It removes the parser as a variable, so you are comparing models rather than comparing serving-stack support.*

BEFORE YOU MOVE ON

A model emits a well-formed JSON tool call as ordinary assistant text, and the client never sees a structured call. What is the fault?

A. The model has not been trained for tool use and is imitating the format from the prompt.

B. The tool definitions were not included in the chat template, so the model invented the format.

C. The serving layer has no parser configured for this model's tool-call format, so the correct output is passed through as content.

D. Constrained decoding is disabled, so the call is not being validated.

46 · 9 min

Agent loops on models that are not clever

THE ONE THING TO KEEP

Agent loops on small models fail by never stopping, repeating actions, inventing observations, losing the goal and outgrowing the context — so hold the state, the deduplication and the step cap in code and reduce the number of model decisions rather than enlarging the model.

The loop, and the five ways it dies

An agent is a loop: the model chooses an action, something executes it, the result goes back into the context, repeat until done. On a frontier model this often works. On an 8B model it fails in five specific ways, and each has a structural fix that does not involve a bigger model.

1. It never decides it is finished. There is no natural stopping signal, so the loop runs to its step limit. *Fix:* make finishing an explicit action with a schema — `{"action":"done","answer":"..."}` — and cap the steps. Never rely on the model producing a sentence you pattern-match on.

2. It repeats the same action. The model searches for the same string four times, because each turn it sees a context in which searching seemed reasonable. *Fix:* keep the history of attempted actions outside the model and refuse duplicates in code, returning `already tried, result was X` as the observation.

3. It hallucinates the result. Rather than calling the tool, it writes what the tool would probably have returned and proceeds. This is common and dangerous because everything downstream looks fine. *Fix:* only ever advance the loop on results your code produced. Ignore any observation text the model wrote itself.

4. It loses the goal. By step six the original instruction is buried under observations, and recency wins. *Fix:* restate the goal in every turn, immediately

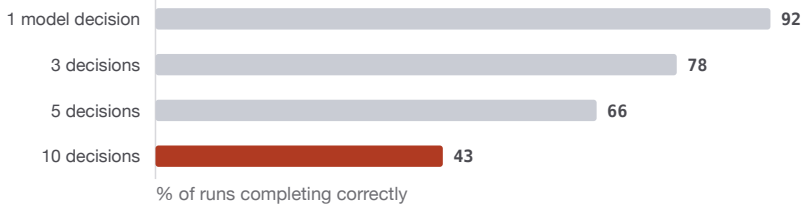
before the model generates. It costs fifty tokens and it is the highest-value line in the loop.

5. Context grows until it breaks. Each step adds an observation; long tool outputs add a lot. Ten steps can exhaust an 8k window. *Fix:* truncate observations to what matters, and keep a running structured state — a small JSON object of what has been established — rather than the raw transcript.

The arithmetic that should change your design

Compounding is the reason all of this matters. At 92% per-step reliability, a ten-step loop completes correctly 43% of the time. Getting to 90% overall from ten steps requires 99% per step, which no small model provides on an open-ended decision.

A loop at 92 per cent per-step reliability



Ten steps complete correctly forty-three times in a hundred. Reaching ninety per cent over ten steps needs ninety-nine per cent per step, which no small model gives on an open-ended decision — so remove decisions rather than enlarge the model, and verify after each one that remains.

There are only two ways out, and increasing the model size is the weaker one.

Reduce the number of model decisions. Most of what is written as an agent loop is a workflow with a fixed shape: fetch, extract, decide, write. Encode the shape in code and use the model at the two or three points that genuinely need judgement. Five deterministic steps with two model calls at 92% is 85% overall, not 43%.

Verify after each step. A check that catches a bad action and retries turns a 92% step into an effective 99% step, and it is usually a schema validation or a lookup rather than another model call.

What a working small-model agent looks like

```
state = {"goal": goal, "facts": {}, "tried": []}
for step in range(MAX_STEPS):
    prompt = render(state)                          # goal re-
stated last
    action = model_json(prompt, schema=ACTION)    # constrained
decoding
    if action["type"] == "done":
        break
    if action in state["tried"]:
        state["facts"]["note"] = "repeated action refused"
        continue
    state["tried"].append(action)
    obs = execute(action)                           # your code,
always
    state["facts"][action["type"]] = summarise(obs,
max_chars=600)
```

Everything that matters is outside the model: the state, the deduplication, the truncation, the step cap, the schema. The model contributes one constrained decision per iteration.

Where small models genuinely do well

Narrow loops with few options and verifiable results. Retrieve and answer with citations. Fill a form by looking up three fields. Triage an item into one of five queues, calling one lookup. Retry a failing structured extraction with the error message as feedback.

Where they do not: open-ended research, multi-hour tasks, anything where the correct next step depends on judgement about a situation the model cannot verify.

> *The honest framing for anyone planning work: a well-built pipeline with an 8B model at three decision points is more useful and far cheaper than a free-form agent with a 70B, and it is debuggable. Free-form autonomy is the most expensive way to buy reliability, at any model size.*

BEFORE YOU MOVE ON

A five-step agent using an 8B model succeeds on about 60% of tasks. The team considers moving to a 70B, which tests at 97% per step. What does the arithmetic suggest?

A. The 70B would give about 86% overall, so the real gain comes from removing model decisions or verifying each step rather than from the larger model alone.

B. The 70B would reach roughly 97% overall, since per-step accuracy is the dominant factor in a short loop.

C. Success rates do not compound in agent loops because later steps can correct earlier errors.

D. The 60% figure implies per-step accuracy around 60%, so the model is much weaker than tested.

47 · 9 min

What these models really do in your language

THE ONE THING TO KEEP

Language support divides into well-supported, thin and nominal tiers with a steep cliff between the last two, script and register matter as much as language, and only a native speaker rating thirty real inputs will tell you which tier you are in.

Three tiers, and where the cliff is

Model cards list languages. The list conflates three very different states.

Well supported. English, and to varying degrees Chinese, Spanish, French, German, Portuguese and Russian. Fluent, idiomatic, reliable instruction following, reasonable factual coverage.

Present but thinner. Hindi, Arabic, Bengali, Indonesian, Vietnamese, Turkish, Japanese, Korean and others. Fluent-sounding output with more factual error, weaker instruction adherence, occasional grammatical oddity, and a noticeable drop on anything requiring reasoning in the language rather than translation into it.

Nominal. Most of the world's languages. The model produces something, and a native speaker will find it stilted, wrong, or written in the wrong register. The list on the model card is often derived from what appeared in training data at any volume, not from an evaluation.

The cliff between the second and third tier is steep, and it is where most of the disappointment lives.

Script and register, not just language

Two details specific to how these models are trained, and both matter in South Asia in particular.

Script changes everything. Hindi written in Devanagari and Hindi written in Roman letters are, to the tokenizer and often to the model, different problems. A model may be decent at one and poor at the other depending on what its corpus contained — web forums are heavily romanised, formal publishing is not. Test both scripts separately.

Code-switching is normal and is tested by nobody. Real Hinglish mixes English words into Hindi grammar, in Roman script, with English technical vocabulary. Models handle this unevenly and no public benchmark covers it. If your users write that way, your evaluation set must too, and it will not resemble anything you can download.

Register. A model may produce grammatically correct Hindi that reads as translated newspaper prose to somebody who would have written it conversationally. This is invisible to automated metrics and obvious to a native speaker in one sentence.

The translation sandwich

A common strategy: translate the input to English, do the work, translate back. It works, and it has three costs worth naming.

- **Two extra model calls** of latency and expense.
- **Culture-specific content degrades.** Names, honorifics, local references, legal and administrative terms with no English equivalent.
- **Errors compound across three stages**, and the failure appears in the final output where it is hardest to attribute.

It remains the right answer when the task is genuinely language-neutral — classification into English labels, extracting a date — and the wrong answer when the output goes to a user in their language.

A better version when you can afford it: do the work in the original language with a model that supports it, and use translation only to let *you* check the output.

Measuring, since benchmarks will not

No public benchmark will tell you whether a model is good enough for your users in Marathi. The procedure that will, and it takes about two hours:

1. Collect **30 real inputs** in each language you care about, in the script your users actually use.
2. Write what a good response looks like, or a one-line rubric.
3. Run three candidate models at temperature 0.
4. Have a **native speaker** rate each output on three points: acceptable, awkward but usable, wrong.
5. Count tokens per input for each model while you are there.

The token count is the cost decision and the ratings are the quality decision, and neither is available anywhere else. Repeat it when you change model, quantisation or provider, because quantisation damages lower-resource languages more than English, as an earlier lesson showed.

What to reach for when the answer is not good enough

- **A language-focused model.** The families in the earlier lesson exist for this and often win decisively.
- **Fine-tuning on a few hundred examples in your language and register.** This fixes register and format better than it fixes knowledge, which is usually what is wrong.
- **Retrieval in the user's language.** Much of what looks like a language weakness is a knowledge weakness, and giving the model the right paragraph in the right language fixes both.
- **A larger model.** Multilingual quality scales with size more steeply than English quality does, so this helps more here than elsewhere — at the cost the earlier lessons described.

> *The habit that matters most: never accept a language claim without a speaker of that language reading twenty outputs. Fluency and correctness diverge sharply outside the first tier, and fluency is what fools a non-speaker.*

BEFORE YOU MOVE ON

A team evaluating Hindi support finds a model scores well on a public Hindi benchmark but users say the output reads oddly. What is the most likely explanation?

A. The benchmark was contaminated, so the score is meaningless.

B. The model is producing correct but stilted, translation-flavoured Hindi in the wrong register and possibly the wrong script, which automated scoring does not capture.

C. Quantisation has damaged the Hindi capability specifically, and the benchmark was run at full precision.

D. The tokenizer is splitting Devanagari inefficiently, which degrades output quality.

48 · 9 min

Refusals, and the uncensored question

THE ONE THING TO KEEP

Refusal is generated behaviour mediated by a direction in activation space that can be removed from open weights in an afternoon, so a boundary your product needs must be enforced outside the model rather than by its tuning.

Refusal is a behaviour, not a filter

An open model has no content filter. Nothing inspects the output and blocks it. When a model declines a request, it is *generating* a refusal, because preference tuning made refusal the likeliest continuation for inputs shaped like that one.

That distinction has a mechanical consequence. Interpretability research has found that refusal behaviour in these models is mediated substantially by a single direction in the model's activation space — a direction that can be identified by comparing activations on harmful and harmless prompts. Subtract that direction from the weights and the model stops refusing, without retraining. The technique is called abliteration, and modified models produced this way are widely published.

This is worth understanding rather than moralising about, because it settles an argument. **Safety tuning on open weights is not a control.** It is a default. Anyone with a GPU and an afternoon can remove it, and many have. Whatever protection open release provides, it does not come from the tuning surviving contact with a determined user.

What abliteration costs

It is not free, and the costs are measurable.

- **General capability drops.** The intervention is blunt; the direction it removes is entangled with other behaviour. Modified models usually score somewhat worse on ordinary benchmarks than their originals.
- **The model loses the ability to refuse anything.** Including the things you would want refused — a support bot that will now enthusiastically help with fraud, a children's product with no floor.
- **Provenance disappears.** Most abliterated uploads have no evaluation, no dataset statement, and no accountable maintainer. Everything in the vetting lesson applies twice over.

Over-refusal is the more common product problem

For most people building things, the practical difficulty is not that the model will not help with something harmful. It is that it refuses things it should not:

- A support message mentioning self-harm gets a refusal instead of an escalation, which is the worst possible response.
- Medical, legal or security content is declined even in a professional context.
- Fiction involving violence is refused.
- A translation is declined because the source text contains a slur.

This is a real and measurable defect, and the first thing to do is measure it. Build a **false-refusal set**: 30 inputs that are legitimate for your product and sit near the boundary. Count refusals. Track it like any other metric, because a model swap can change it sharply in either direction.

The ladder, before you reach for a modified model

1. **A system prompt that establishes the context.** Naming the professional setting resolves a large fraction of over-refusals, and it costs nothing.
2. **A different model.** Refusal thresholds vary widely across families for identical inputs. Test three; the spread will surprise you.
3. **A base model.** Base models refuse nothing because they follow nothing. If you are fine-tuning anyway, starting from base gives you control over re-

fusal behaviour as part of your own tuning.

4. **Your own tuning.** A few hundred examples of the behaviour you want, including refusals of the things you *do* want refused. This is the defensible route: you get a model with a refusal policy you wrote and can explain.
5. **An abliterated model**, understanding that you now have no floor and no provenance.

Most teams that end at step 5 needed step 1.

The unresolved policy question, stated fairly

Whether open release of capable models increases real-world harm is genuinely disputed. One side argues that safety tuning is trivially removable, so open weights hand capability to anyone. The other argues that the marginal uplift over existing resources is small for most harms, and that open weights bring large offsetting benefits in transparency, competition and independent safety research.

The empirical work so far does not settle it, and the honest position is that this depends on the capability in question and that reasonable people who have read the same evidence disagree. What is *not* in dispute is the mechanics: the tuning can be removed cheaply, and any argument that relies on it holding is unsound.

> *For your own work the practical conclusion is narrower. If your product needs a boundary, enforce it outside the model, with a separate check that the model cannot talk its way past. That is the next lesson.*

BEFORE YOU MOVE ON

A team wants to guarantee their open-weights product never produces a particular category of content. Which approach can actually provide that guarantee?

A. Choosing a model whose safety tuning is strongest for that category, verified on a test set.

B. Fine-tuning the model on examples of refusing that category.

C. An external check on the output that the model cannot influence, which blocks or replaces responses independently of what the model generated.

D. A system prompt forbidding the category, placed at the end of the prompt where adherence is strongest.

49 · 8 min

Guardrails as a separate layer

THE ONE THING TO KEEP

A separate classifier is a different failure surface from the generating model because it labels rather than obeys, so layer cheap deterministic checks, a small input classifier, the model and an output check — and never let any of it stand between a model and an irreversible action.

Why the check must not be the same model

A model asked to both answer the user and police itself has a structural problem: the thing it is policing is in its own context, written by the person it is policing. Prompt injection works precisely because instructions and data arrive through the same channel.

A separate classifier does not have that problem to the same degree. It is not following instructions; it is producing a label. Text saying *ignore your previous instructions* is an input to be classified, not a command, because the model was tuned to classify rather than to comply.

That is not perfect either — classifiers can be evaded — but it is a genuinely different failure surface, and layering the two is much stronger than either alone.

The open guardrail models

Llama Guard classifies against a hazard taxonomy — violent crime, sexual content, self-harm, privacy, intellectual property and others — and runs on both the user's input and the model's output. You can edit the taxonomy in the prompt, which is more useful than it sounds: the categories you care about are rarely exactly the default list.

ShieldGemma does the same job in the Gemma family, in several sizes including very small ones.

Granite Guardian from IBM covers harms plus groundedness and relevance checks for retrieval systems.

Prompt-injection classifiers are typically small encoder models — DeBERTa-sized, tens of millions of parameters — trained to spot injection attempts. They run in single-digit milliseconds on a CPU, which is what makes them practical on every request.

NeMo Guardrails is orchestration rather than a model: a framework for defining conversational rails and wiring checks around a model.

Order them by cost

The pipeline that works is cheapest-first, so the expensive checks only see what survives.

1. **Deterministic checks.** Length limits, allow-lists, regular expressions for obvious patterns, rate limits per user. Microseconds, and they catch a surprising amount.
2. **A small classifier on the input.** Injection detection and hazard classification. Single-digit milliseconds on a CPU.
3. **The model.**

4. **A check on the output.** Hazard classification again, plus whatever your domain requires — a groundedness check for retrieval, a personal-data detector, a schema validation.

Step 4 matters more than people expect. Plenty of inputs look benign and produce output you cannot ship. Checking only the input is the more common design and the weaker one.

The cost, and how to pay less of it

A guardrail model adds latency to every request. A small classifier at 20 milliseconds against a 2-second generation is 1% and invisible. A large guardrail model doing a full generation to produce a verdict can add hundreds of milliseconds twice, which is a product decision rather than a technical one.

Two mitigations. **Run the output check concurrently with streaming** and be prepared to withdraw the response — this is what most production systems do. And **use the smallest classifier that hits your target**, measured on your own inputs rather than on the published numbers.

False positives are the expensive failure

A guardrail that blocks legitimate requests fails your users silently. They do not report it; they leave. Measure both directions, on your own data:

- **False negatives:** attempts that got through. Collect them from red-teaming and from incidents.
- **False positives:** the false-refusal set from the previous lesson, run through the guardrail rather than the model.

Report both numbers together and tune the threshold to a ratio you chose deliberately, not to whatever the default was.

What no guardrail gives you

Robustness against a determined adversary. Every published classifier has been evaded, and evasion techniques circulate quickly. Layered defence lowers the rate of successful attacks; it does not make the risk zero.

The design consequence is architectural rather than model-shaped: **do not put a language model where a successful attack causes irreversible harm.** No amount of classification makes it safe to let a model, on its own, send money, delete data, or publish to the world. Keep a person or a deterministic rule in the path of any action you cannot undo.

BEFORE YOU MOVE ON

A team adds a guardrail model that checks user input only, and later finds harmful content in responses even though every input was benign. What does this show?

A. The guardrail model is too small and needs replacing with a larger one.

B. The input classifier was evaded by an injection technique it had not seen.

C. Benign inputs can produce unshippable output, so the output side needs its own check regardless of how good the input classifier is.

D. The generating model's own safety tuning has degraded, probably through quantisation.

50 · 9 min

When the model does not know

THE ONE THING TO KEEP

A model always produces its most likely continuation, so abstention must be made available in the schema and rewarded in the prompt, and requiring a verifiable quotation turns invisible fabrication into a check that code can fail.

There is no empty answer

A language model produces the most likely continuation of its context. There is no state in which it produces nothing. If the answer is not available, the

most likely continuation is a plausible-looking answer, because that is what text answering a question looks like.

This is why *hallucination* is a slightly misleading word. The model is not malfunctioning; it is doing exactly what it does, on an input where the correct behaviour would have been to stop. Saying *I do not know* is a specific behaviour that has to be trained in and then made likely by the prompt and the format, and small models have less of it.

Confidence is only weakly informative

The obvious idea is to read the model's own probabilities. It half works.

Token probabilities correlate with correctness, but weakly and inconsistently, and instruction tuning makes it worse — preference tuning rewards confident-sounding answers, which pushes probabilities up regardless of correctness. A model can assign 0.94 to a fabricated citation.

Where logprobs are genuinely useful is on **constrained** outputs. If the model must choose between five enum values, the probability mass over those five is meaningful, and a top choice at 0.4 against a runner-up at 0.35 is a real signal that this item needs a human. For free-form text, treat logprobs as a weak prior, not a confidence score.

Five things that work, in order of return

1. Give it the information. Most confabulation is a knowledge problem, and retrieval solves knowledge problems that no model size solves reliably. A 4B model with the right paragraph beats a 70B model guessing, and it is cheaper.

2. Make abstention available in the format. If the schema requires an answer, the model must invent one. Give it somewhere to go:

```
{"answer": {"type": ["string","null"]},
 "evidence_quote": {"type": ["string","null"]},
 "status": {"enum": ["answered","not_in_source","ambiguous"]}}
```

Then say plainly in the prompt that `not_in_source` is a correct and expected answer. Small models need explicit permission; without it, abstention has essentially zero probability.

3. Require a quotation. Ask for the exact span from the source that supports the answer, and verify in code that the span really appears in the source. This is a hard check, not a soft one — a fabricated quote fails a string search — and it converts an invisible failure into a caught one. It is the single most effective technique in this list after retrieval.

4. Self-consistency. Sample five completions at temperature 0.7 and compare. Agreement across samples correlates with correctness considerably better than any single answer's confidence. Disagreement is a reliable flag for review. It costs five times as much, so apply it to the items that matter rather than to everything.

5. A verification pass. A second call, or a second model, given the source and the proposed answer and asked only whether the source supports it. Cheaper than it sounds, because the verification prompt is short and the answer is one token.

Measure it, with an unanswerable set

Build a small set that makes fabrication visible:

- 20 questions whose answers **are** in your documents.
- 20 questions that **look** answerable but are not — a plausible field that the document does not contain, a person not mentioned, a date not given.

Score two numbers separately: accuracy on the answerable half, and abstention rate on the unanswerable half. A model at 90% accuracy that abstains on 10% of the unanswerable questions is worse for most products than one at 82% accuracy that abstains on 85%, because the second one's errors are visible.

Two systems on twenty answerable and twenty unanswerable questions

	Accuracy on the answerable 20	Abstention on the unanswerable 20
System A	**90%** 18 of 20 right	**10%** 18 confident inventions
System B	**82%** 16 of 20 right	**85%** 3 inventions

System A is eight points better on the questions it can answer and produces eighteen confident inventions on the ones it cannot. The second column is the number nobody measures, and it is the one that decides whether people can trust the system.

This second number is the one nobody measures and the one that decides whether people can trust the system.

Say it in the interface

The final piece is not technical. If the system can be wrong, the interface should show its source and make checking easy — the quotation, a link, the document name. A confident sentence with no provenance invites trust the system has not earned; the same sentence with the supporting line beside it invites the reader to do the one check that catches the failure.

BEFORE YOU MOVE ON

Two configurations of a document question-answering system: A answers 90% of answerable questions correctly and abstains on 10% of unanswerable ones; B answers 82% correctly and abstains on 85% of unanswerable ones. Which is usually better in production, and why?

A. A, because accuracy on answerable questions is the metric users experience.

B. B, because its failures mostly surface as an admission of not knowing rather than as confident fabrications the reader cannot detect.

C. A, because B's high abstention rate means it will also abstain on answerable questions and frustrate users.

D. Neither; both need a confidence threshold on token probabilities to decide when to answer.

CASE STUDY · MODULE 5

The classifier that refused the messages it existed for

A student-support helpline at a university in Hyderabad, building a triage step that routes incoming messages to the right counsellor within minutes rather than hours.

The helpline received around four hundred messages a week, rising to twelve hundred in the fortnight around end-semester examinations. Four counsellors read everything in arrival order. The project had one purpose: read each message as it arrived, put it in one of five queues, and push anything in the urgent queue to a counsellor's phone immediately rather than at the top of the next hour.

Nothing was to be answered by a model. The output was a queue name and a confidence, and a person read every message regardless. Lakshmi, the postgraduate student building it under the counselling head's supervision, had been explicit about that in the one-page design, because the first question from the ethics committee had been whether a machine would be replying to a distressed student, and the answer was no.

The failure that appeared on the second day of testing

She ran a hundred and twenty real messages, anonymised, through an 8B instruct model with a clear classification prompt. It worked on ninety-odd of them.

On the messages that mentioned self-harm, it refused. Not a wrong label — a refusal: a paragraph declining to engage and suggesting the person contact a professional. The model was being asked to label a message, and it was reading the message as a request and declining it.

Those messages are the entire reason the urgent queue exists. A triage step that fails exactly on the inputs it was built to catch is worse than no triage step, because the system is now unreliable in a way that correlates with severity. In the arrival-order system, a message mentioning self-harm was read by a

counsellor within the hour. In the new system it would fall out of the classifier and, unless somebody had thought about it, land nowhere.

What was available, and what it cost

A modified model. A search of the public hub turned up several copies of the same base with refusal behaviour stripped out. They were free and available that afternoon. They also had no evaluation, no dataset statement and no accountable maintainer, and they refuse nothing at all, which in a product used by distressed students is not a neutral property. The counselling head's position was that she could not explain that choice to a parent, and that was the end of it as a serious option.

Context in the system prompt. Lakshmi tried it: a system message naming the professional setting, stating that the output is a routing label read by a qualified counsellor, and that the message is being categorised rather than answered. Refusals on the self-harm messages fell from every one to about one in six. Cost: twenty minutes.

A different model. She ran the same hundred and twenty through three other open models of similar size. The refusal rates on the same messages were wildly different — one family refused twice as often as another on identical inputs. Cost: an afternoon, and it identified a model that refused four of the relevant messages rather than nineteen.

Constrain the output. Force the answer to be one of five queue names plus an `unclear` value, so a refusal paragraph is not a token sequence the model can produce. Cost: an hour. This removed the failure mode entirely for the classifier, though it did not remove the underlying tendency — it made it come out as `unclear` instead of as prose.

Their own tuning. Three hundred labelled messages from the counsellors, a small adapter, and a model with a routing behaviour the university had written and could explain. Cost: a counsellor's time to label, two to three weeks, and it would land after the examination peak.

The decision

The peak was five weeks away and the counselling head had to choose what ran during it. The argument that decided it came from one of the counsellors, who pointed out that the question was not which model to trust but what happens to a message the system is unsure about. If `unclear` went to the front of a human queue rather than the back, the worst case of the new system was the old system.

> **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 shipped the boring stack and it held through the peak. A system prompt naming the setting, the model that refused least of the three tested, grammar-constrained output over six values, and a rule that `unclear` and the urgent queue both alerted a counsellor immediately. A separate small input classifier ran ahead of the model on every message and could raise the urgent flag on its own, so the routing decision never depended on a single model's cooperation. Lakshmi also built a thirty-item false-refusal set — legitimate messages sitting near the boundary — and it became a standing check: when they swapped the model in the following term, the refusal rate on those thirty moved sharply, and they caught it before deployment rather than after. The fine-tune on counsellor-labelled messages happened in the following term, unhurried, and the abliterated model was never used.

The model refused a classification task. Mechanically, what is happening, and why does that explain why constrained output fixed it?

An open model has no filter inspecting its output; a refusal is generated text, because preference tuning made refusal the likeliest continuation for inputs shaped like that one. Grammar-constrained decoding masks every token that is not part of a legal answer at each step, so the refusal paragraph is not merely unlikely, it is im-

possible to emit. The model's underlying tendency has not changed — it now surfaces as an `unclear` label instead — which is why the design still needed a rule for what happens to `unclear`.

Why was the abliterated model a bad answer here even though it would have solved the refusal problem in an afternoon?

Because it removes the ability to refuse anything, not just the refusals that were wrong, and it arrives with no evaluation, no dataset statement and no maintainer, so every vetting concern applies twice. A boundary the product needs has to be enforced outside the model, by a separate classifier and by deterministic rules, because safety tuning on open weights is a default rather than a control. Reaching for a stripped model here would have traded a visible, fixable failure for an invisible, unbounded one in a service used by distressed students.

The counsellor's argument was about what happens to an uncertain message rather than about model choice. Why is that the stronger framing?

Because it makes the system's worst case explicit and bounded. If `unclear` goes to the front of a human queue, the new system degrades to the old system rather than to silence, and no model failure can make things worse than they were. It also puts the design decision where it belongs: the model is one component with a known failure rate, and the safety of the service comes from the routing around it — never from the model being reliable enough on its own.

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 A seven-billion model produces valid JSON for the first two fields and then breaks the syntax, under an interface's default settings. Which default is the likeliest cause?

A. A repetition penalty above one, which suppresses the punctuation JSON repeats

B. Temperature at 0.8, which makes the model invent field names not in the schema

C. top_k at 40, which is too small to contain every legal JSON token at each step

D. top_p at 0.95, which admits tokens outside the schema's vocabulary

2 After switching to schema-constrained decoding, parse failures fall to zero, but an audit finds invented amounts on documents that contained none. What went wrong?

A. Constrained decoding raises the temperature internally to keep the grammar satisfiable

B. The model was too small to hold the schema and the document at the same time

C. The grammar compiler mis-parsed the schema and dropped the required list

D. A required field with no null or unknown value is an instruction to fabricate

3 A four-billion model keeps mentioning the price despite an instruction not to. Which change is most likely to work?

A. Move to a larger model, because suppression needs more reasoning capacity
B. Repeat the prohibition three times at the top of the prompt
C. State positively what to discuss, placed after the input
D. Raise the temperature so the model is less anchored on the forbidden word

4 Research on in-context learning found that corrupting the labels in few-shot examples hurt far less than removing the examples altogether. What should that change about how you build an example set?

A. Use fewer examples, since the labels evidently contribute very little
B. Spend the effort on consistent formatting and covering every label
C. Use deliberately wrong labels to stop the model copying the answers
D. Replace the examples with a longer written description, which carries the mapping better

5 An eight-billion model in an agent loop sometimes writes what a tool would have returned and carries on from it. Which structural fix removes the failure?

A. Only advance the loop on results your own code produced
B. Raise the step cap so the model has room to call the tool properly
C. Move to a reasoning model, which checks its own observations before proceeding
D. Ask the model to confirm in its reply that it really called the tool

6 A retrieval system answers every question confidently, including ones its documents do not cover. Which change most directly turns invisible fabrication into a check code can fail?

A. Raise the number of retrieved chunks so the answer is more likely to be present

B. Ask the model to rate its own confidence from one to five in every answer

C. Read the model's token probabilities and reject answers below a threshold

D. Require an exact quotation and verify it appears in the source

MODULE 6

Adapting a model to your work

Fine-tuning is the freedom that open weights actually deliver, and it is also the most over-reached-for tool in the field. This block gives the decision rule for when it is warranted, the arithmetic of LoRA and QLoRA, how to build a dataset that does not encode your own inconsistency, a run you can complete on free hardware, preference tuning, distillation, model merging, the forgetting you have to measure, and how to ship the result.

> BY THE END OF THIS MODULE YOU CAN
>
> Decide with evidence whether a task needs fine-tuning at all; if it does, build the dataset, size a LoRA to fit free hardware, run it, and demonstrate on a held-out set and a regression suite that the adapted model is better at the task and no worse at everything else

51 · 8 min

When not to fine-tune

THE ONE THING TO KEEP

Fine-tuning reliably changes form and unreliably changes knowledge, so the honest test is whether it beats your best prompt plus retrieval by a margin that justifies re-tuning every time the base model is superseded.

What fine-tuning changes, and what it does not

Fine-tuning adjusts weights on examples of the behaviour you want. It is very good at some things and unreliable at others, and the difference decides whether you should do it.

It reliably changes form. Output format, tone, register, verbosity, structure, the conventions of your domain, which of several plausible responses is preferred, and whether the model does the task at all rather than explaining it. This is what fine-tuning is for.

It changes knowledge poorly. You can teach facts by fine-tuning, and it works badly compared with putting the facts in the context. The model learns the surface of the statement, generalises it unpredictably, contradicts it when phrased differently, and forgets other things in the process. Retrieval is better on every axis: cheaper, updatable, citable, and it does not degrade anything else.

It does not add reasoning capacity. A 3B model tuned on your data is still a 3B model. Tuning can make it apply what it has more reliably to your task; it cannot make it think further ahead.

The ladder, in order of cost

Work down this list and stop when the result is good enough.

1. **A better prompt.** An hour, and it fixes more than people expect.

2. **Three to eight examples in the prompt.** Another hour.
3. **Constrained output.** Removes format failures entirely.
4. **Retrieval.** Fixes knowledge problems, which is most of what looks like model weakness.
5. **A different or larger model.** A day of evaluation.
6. **Fine-tuning.** Days of data work, then permanent maintenance.

Most teams that fine-tune skipped steps 3 and 4. That is the single most common waste in this part of the field.

The signals that fine-tuning is genuinely right

- **You have, or can get, 300 or more real examples** of exactly the input and the output you want. Below that, few-shot prompting is usually as good.
- **A long prompt is being paid on every request.** If you are sending 2,000 tokens of instructions and examples a hundred thousand times a month, that is 200 million input tokens. Tuning that behaviour into the weights removes the recurring cost, and this is often the strongest financial argument.
- **You need a smaller model than the task needs.** A tuned 3B that matches an untuned 30B on your narrow task costs an order of magnitude less to serve and runs where the 30B cannot. This is the main reason fine-tuning is worth it on device.
- **A format or style must hold over long outputs.** Prompted constraints decay; trained behaviour does not.
- **The model is systematically wrong in a way you can demonstrate**, on a domain with its own conventions, and you have examples of right.

The cost people underestimate

The run is the cheap part. A LoRA on 500 examples is under an hour on a free notebook GPU.

The expensive parts are the dataset, which is two or three days of careful work by somebody who understands the task, and the **permanence**. A fine-tune is a fork. When a better base model appears in four months, you re-tune, re-evaluate and re-deploy, or you stay on the old base while the field moves. Every fine-tune is a recurring commitment, and teams routinely take on three or four of them before noticing.

Budget it honestly: the first tune costs days, and each subsequent base upgrade costs a day. If the improvement does not justify that annually, do not start.

A decision you can defend

Write down, before you begin, three things:

1. The number your fine-tune must beat, measured on a held-out set with the best prompt you could write.
2. The margin that would count as worth it, given the noise on a set that size.
3. What you will do when the base model is superseded.

If you cannot fill in the third line, you are not ready to fine-tune. That is not a rhetorical flourish; abandoned fine-tunes running on two-year-old bases are one of the most common forms of technical debt in this work.

> *The most useful sentence in this lesson, if you take only one: measure the best prompted result first, write it down, and treat it as the thing to beat. Half the time nothing beats it and you have saved a week.*

BEFORE YOU MOVE ON

A team's model gets product specifications wrong. They plan to fine-tune on 2,000 question-and-answer pairs drawn from their specification documents. What is the strongest objection?

A. 2,000 examples is too few to teach a model anything, so they need at least ten times more.

B. This is a knowledge problem, which fine-tuning solves poorly and unpredictably, while retrieval solves it cheaply, updatably and with citations.

C. Fine-tuning on question-and-answer pairs will damage the model's general instruction following.

D. They should use a larger model, since specification detail requires more parameters.

52 · 9 min

What a LoRA actually is

THE ONE THING TO KEEP

LoRA freezes the base and trains a low-rank product added to each targeted matrix, so under 1% of parameters carry gradients — and rank, alpha and which modules are targeted are the three settings that decide whether the run does anything.

The trick, in one equation

Full fine-tuning updates every weight matrix `W` by some `ΔW` of the same shape. LoRA observes that the useful part of `ΔW` is empirically low-rank, and represents it as a product of two thin matrices:

```
W' = W + (alpha / r) * B @ A

W : d x k   (frozen, never updated)
B : d x r   (initialised to zeros)
A : r x k   (initialised randomly)
r : the rank, typically 8 to 64
```

The base weights never move. Only `A` and `B` are trained. Because `B` starts at zero, the model begins identical to the base and departs from it gradually, which is why LoRA training is stable.

The arithmetic that makes it cheap

Take one 4096 × 4096 projection. Full tuning updates 16.8 million parameters. At rank 16, LoRA trains 4096 × 16 plus 16 × 4096 = **131,072** parameters, which is 0.8% of the matrix.

Across a whole 8B model, targeting all the linear layers at rank 16 gives roughly 20 to 40 million trainable parameters — half a percent of the model. Those are the only parameters needing gradients and optimiser state, which is where the memory saving comes from, and it is why a fine-tune that would need several datacentre GPUs fits on a free notebook.

The saved adapter is 40 to 200 MB, against 16 GB for a full copy. You can keep fifty of them.

The same tune, two budgets

Full fine-tuning an 8B

- Every weight updated
- Gradients and optimiser state for 8 billion
- Around 110 GB of training memory
- A 16 GB copy of the model per variant
- Several datacentre GPUs

LoRA at rank 16, all linear layers

- Base frozen, and B starts at zero
- About 40 million trainable, half a per cent
- Fits on a free notebook GPU
- A 40 to 200 MB adapter per variant
- Keep fifty of them

One 4096 by 4096 projection is 16.8 million parameters fully tuned, or 131,072 at rank 16. Print the trainable percentage on every run: if it is far from what you expected, the target modules are wrong and the next three hours will produce nothing.

The three settings that matter

Rank `r`. The capacity of the update.

- 8 to 16: format, tone, style, output structure. Most tasks are here.
- 32 to 64: a new task, a domain with its own vocabulary, a substantial behavioural shift.
- 128 and above: approaching full fine-tuning in capacity, and usually a sign that continued pretraining is the real requirement.

Higher rank is not safer. It has more capacity to overfit a small dataset and it forgets more.

Alpha. A scaling factor: the update is multiplied by `alpha / r`. The convention `alpha = 2 * r` is a reasonable default. What matters is that alpha and rank move together — raising rank while keeping alpha fixed quietly reduces the effective strength of your update, which is a common cause of a tune that seems to do nothing.

Target modules. Which matrices get adapters. The original paper adapted only the query and value projections. Later practice found that targeting **all linear layers** — query, key, value, output, and the three feed-forward matrices — works measurably better for the same parameter budget, because most of the model's parameters are in the feed-forward stack.

```
from peft import LoraConfig, get_peft_model
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="CAUSAL_LM",
)
model = get_peft_model(base, cfg)
model.print_trainable_parameters()   # trainable: 41,943,040 ||
all: 8,072,204,288 || 0.52%
```

Print that last line every time. If the trainable percentage is far from what you expected, your target modules are wrong, and the run will produce nothing useful over several hours.

Merging, or not

An adapter can be folded into the base weights permanently:

```
merged = model.merge_and_unload()      # W += (alpha/r) B A
merged.save_pretrained("./my-model")
```

Merged models load anywhere and need no adapter support. Unmerged adapters stay tiny, can be swapped at request time, and let one base serve many tunes. The choice belongs to the shipping lesson at the end of this block; for now, keep the adapter, because you can always merge later and cannot easily unmerge.

Why it works at all

The honest answer is empirical. The observation is that fine-tuning updates for a specific task have low intrinsic rank — the model is being nudged within a subspace it already occupies rather than being taught something structurally new. This is consistent with the other findings in this block: LoRA is excellent at form and weak at knowledge, because form is a nudge and knowledge is not.

Where LoRA underperforms full tuning is precisely where you would predict: large amounts of genuinely new material, a new language, a new modality. For those, the answer is not a higher rank; it is continued pretraining, several lessons from here.

BEFORE YOU MOVE ON

A team raises LoRA rank from 16 to 64 to give the adapter more capacity, leaving lora_alpha at 32. The tuned model changes less than before rather than more. Why?

A. Rank 64 overfits the small dataset, so the model reverts towards the base during training.
B. The update is scaled by alpha divided by rank, so holding alpha fixed while raising rank quartered the effective strength of the update.
C. Higher rank requires proportionally more data, so with the same dataset the extra parameters stay near zero.
D. Rank above 32 requires targeting all linear layers, and attention-only targeting caps the achievable change.

53 · 9 min

Fitting a fine-tune in 16 GB

THE ONE THING TO KEEP

Training memory is weights plus gradients plus optimiser state plus activations, and QLoRA collapses the first three by freezing a 4-bit base and training a small adapter — leaving activations, controlled by sequence length, batch size and gradient checkpointing, as the dial you actually turn.

Where the memory goes in training

Inference needs weights plus a cache. Training needs considerably more, and knowing the four components tells you exactly which dial to turn when it will not fit.

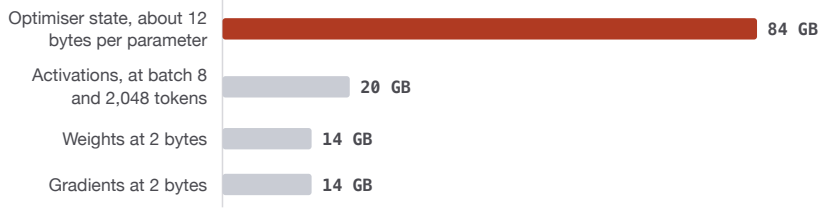
For a full fine-tune of a 7B model in mixed precision:

- **Weights**, 2 bytes each: 14 GB

- **Gradients**, one per weight, 2 bytes: 14 GB
- **Optimiser state**, Adam keeps two moments in 32-bit plus a master copy: roughly 12 bytes per parameter, 84 GB
- **Activations**, everything saved for the backward pass: depends on batch size and sequence length, often tens of GB

That is 110 GB or more, which means several datacentre GPUs. It is why almost nobody full-tunes a 7B model.

Full fine-tune of a 7B: where the memory goes



About 110 GB, which is why almost nobody full-tunes a 7B. QLoRA freezes a 4-bit base at 3.5 GB and trains an adapter, taking the first, third and fourth bars to under five in total — leaving activations, the one bar you control, with sequence length, batch size and gradient checkpointing.

What QLoRA removes

QLoRA combines three ideas and the result fits on a free notebook GPU.

The base is quantised to 4-bit and frozen. Weights drop from 14 GB to about 3.5 GB. Because the base is never updated, quantisation error does not accumulate — it is a fixed distortion the adapter learns against.

Only the adapter trains. With 40 million trainable parameters instead of 7 billion, gradients and optimiser state fall from 98 GB to under a gigabyte.

Paged optimiser state moves optimiser memory to CPU RAM when a spike would otherwise overflow, which turns an occasional crash into an occasional slowdown.

The format is NF4, a 4-bit representation shaped for normally distributed values, plus double quantisation, which quantises the block scales themselves for a further small saving.

New total for a 7B: about 3.5 GB of base, under 1 GB of adapter and optimiser, plus activations. On a 16 GB card that leaves 11 GB for activations, which is the number you now control.

Activations are the dial you actually turn

Activation memory scales with **batch size × sequence length**, roughly linearly in both. When a run fails with an out-of-memory error, this is nearly always what to change.

Gradient checkpointing is the biggest single lever. Instead of storing every intermediate activation for the backward pass, it stores a few and recomputes the rest. Memory falls a lot; compute rises by roughly 30%. Turn it on by default.

Gradient accumulation gives you a large effective batch with a small real one. Batch size 1 with 16 accumulation steps has the same optimisation behaviour as batch size 16 and a sixteenth of the activation memory.

Sequence length is the one people forget. Training at 4,096 tokens when your examples are 600 tokens long wastes memory in proportion. Set `max_seq_length` to your actual data, and use example packing to fill sequences efficiently.

```
from transformers import BitsAndBytesConfig, TrainingArguments
bnb = BitsAndBytesConfig(
    load_in_4bit=True, bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype="bfloat16", bnb_4bit_use_dou-
ble_quant=True,
)
args = TrainingArguments(
    per_device_train_batch_size=1,
    gradient_accumulation_steps=16,      # effective batch 16
    gradient_checkpointing=True,
    optim="paged_adamw_8bit",
    bf16=True, learning_rate=2e-4, num_train_epochs=2,
)
```

What fits where

16 GB (a free notebook T4 or a consumer card): QLoRA on a 7B or 8B at sequence length 1,024 to 2,048. A 13B is tight and works at short sequence lengths.

24 GB (a 3090 or 4090): QLoRA on a 13B comfortably, a 34B at short sequences, an 8B at long sequence lengths.

48 GB: QLoRA on a 70B at modest sequence length. This is where a single rented card becomes interesting.

80 GB: 70B QLoRA with room, or full fine-tuning of a 7B.

The cost in quality

QLoRA against full fine-tuning loses very little on typical instruction-tuning tasks — the published comparisons find the gap small enough to be inside evaluation noise for most work. The base being 4-bit does cost something on tasks that were already fragile at that quantisation, which by the earlier lesson means long context and lower-resource languages.

If you have the memory, `load_in_4bit=False` with a 16-bit frozen base and LoRA adapters is slightly better and still cheap. Use QLoRA because the memory is not there, not out of habit.

> *The practical order when a run will not fit: gradient checkpointing on, sequence length down to your real data, batch size 1 with accumulation, then 4-bit base. Reducing rank is the last thing to try, because it is the only one that reduces what the run can learn.*

BEFORE YOU MOVE ON

A QLoRA run on a 16 GB GPU fails with an out-of-memory error partway through the first epoch, having loaded successfully. Which change addresses the actual cause?

A. Reduce the LoRA rank from 32 to 8, cutting trainable parameters by four.

B. Switch from NF4 to 8-bit quantisation of the base, which is more stable.

C. Enable gradient checkpointing and cut the sequence length to match the data, since activation memory grows with batch size and sequence length.

D. Increase gradient accumulation steps, since more steps spread the memory over time.

54 · 9 min

Five hundred examples, done properly

THE ONE THING TO KEEP

A fine-tune inherits every property of its dataset, so diversity of real inputs, byte-identical output format, measured label agreement and deduplicated splits matter far more than the number of examples.

The dataset is the model

Everything about a fine-tuned model that you did not inherit from the base came from your examples. Every inconsistency in your labels becomes an inconsistency in the model. Every systematic error becomes a systematic behaviour. This is the part of the work that decides the outcome, and it is the part people rush.

How many, and of what

300 to 1,000 examples is enough for most behavioural tuning. Below 200 the run is unstable and few-shot prompting is usually as good; above a few

thousand the returns flatten unless the task is genuinely broad.

Quality dominates quantity at this scale. A careful 400 beats a careless 4,000, because the careless set teaches the model to be careless in the same places.

The format

One conversation per line, as JSONL:

```
{"messages":[{"role":"system","content":"Classify the
ticket."},
             {"role":"user","content":"Card declined twice at
checkout."},
             {"role":"assistant","content":"
{\"label\":\"billing\",\"urgency\":\"high\"}"}]}
```

The trainer applies the model's chat template and, correctly configured, computes loss only on the assistant turn. Check this: training on the loss of the user's tokens teaches the model to generate user messages, and it is a real and quiet defect in hand-rolled training scripts.

Six properties that matter, in order

1. Diversity of inputs. More important than volume. Four hundred examples covering the full range of what arrives beats four thousand variations on the same three shapes. Sample from real traffic across time, source and user type, not from whatever was easy to collect.

2. Consistency of output format. Byte-identical. Same key order, same date format, same capitalisation, same phrasing of the standard sentences. The model copies your format exactly, including the inconsistency.

3. The edge cases you want handled. Empty input, off-topic input, ambiguous input, the case that should be escalated, the thing that should be refused. If none of your 500 examples shows an abstention, the model will never abstain.

4. Real inputs, not invented ones. Invented examples are cleaner, better punctuated and more polite than what people send. A model trained on them

handles real messiness worse, and the gap only appears in production.

5. Label consistency you have measured. Take 30 items, have a second person label them independently, and count agreement. If two careful people agree 80% of the time, the ceiling on your model is around 80%, and no amount of tuning gets past your own disagreement. This measurement takes an hour and reframes the whole project.

6. Splits, held from the start. Training, development and a held-out set you touch once. Deduplicate across them — near-duplicates leaking from training into held-out is the most common reason a fine-tune reports a wonderful number and disappoints in production.

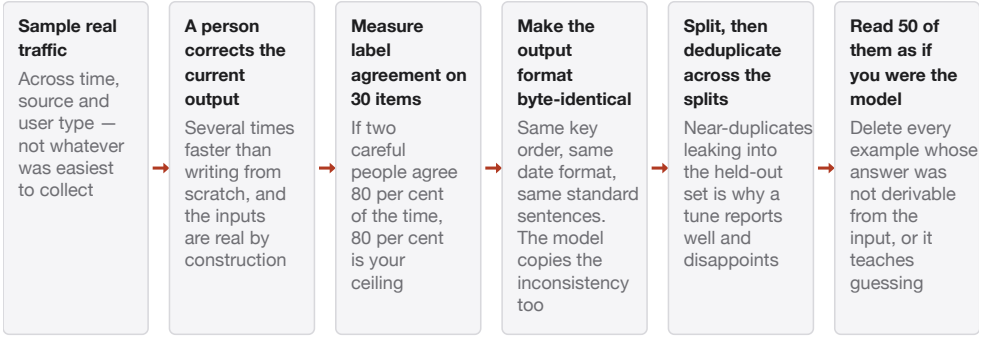
Where the data comes from

Production logs, corrected by a person. The best source by a distance, because the inputs are real by construction. Take the model's current output, have someone fix it, keep the fixed version. Correction is several times faster than writing from scratch.

A larger model, with the terms checked. Generate candidate outputs with a stronger model, then have a human accept, edit or reject each. Two cautions: the terms of service of closed commercial APIs generally prohibit using their outputs to train competing models, and an open-weights teacher passes its own licence conditions to the result. And the student inherits the teacher's style and its errors, so review rather than accept wholesale.

Templating and augmentation. Useful for coverage of rare categories and dangerous as the bulk of a set, because templated variety is not real variety and the model learns the template.

Building five hundred examples in the order that matters



The run is under an hour on a free notebook GPU. This is the two or three days, and it is the part that decides the outcome, because everything the tuned model does that it did not inherit from the base came from here.

The hour that saves the project

Before training, read 50 of your examples end to end as if you were the model. Ask of each: is the output actually derivable from the input? A surprising number of hand-built datasets contain examples where the correct answer depended on information the model was never given — a system the annotator could see, a convention they knew, context from the previous ticket.

Every such example teaches the model to guess. Remove them or add the missing context, and the model gets better at the whole task, not just those items.

BEFORE YOU MOVE ON

Two careful annotators independently label 30 items from a fine-tuning set and agree on 22 of them. What does this tell you before any training happens?

A. The annotators need better training, and once they agree, the model will match their accuracy.

B. Agreement around 73% caps what the model can be measured to achieve, because the labels it learns from and is scored against contain that much disagreement.

C. The sample of 30 is too small to conclude anything, so proceed and measure the model instead.

D. The task is too subjective for fine-tuning and needs a preference-based approach instead.

55 · 10 min

A real run, on free hardware

THE ONE THING TO KEEP

A LoRA run is thirty lines and an hour on free hardware, but the loss curve is only diagnostics — the result is the task metric on a held-out set compared against the base model with your best prompt, at the same settings.

The tools

Unsloth — patched kernels giving roughly twice the speed and noticeably less memory than the baseline, with free notebooks for every popular base model. The fastest path from nothing to a trained adapter.

TRL — Hugging Face's training library. `SFTTrainer` is the standard implementation and what most other tools wrap.

Axolotl — configuration by YAML rather than code. Better for reproducibility and for running many variants.

LLaMA-Factory — a graphical interface over the same ideas, if you would rather not write the script.

All four do the same thing. Pick one and learn its defaults.

A complete run

```
from datasets import load_dataset
from trl import SFTTrainer, SFTConfig
from peft import LoraConfig
from transformers import AutoModelForCausalLM, AutoTokenizer,
BitsAndBytesConfig

base = "Qwen/Qwen3-8B"
tok  = AutoTokenizer.from_pretrained(base)
model = AutoModelForCausalLM.from_pretrained(
    base, device_map="auto", torch_dtype="bfloat16",
    quantization_config=BitsAndBytesConfig(load_in_4bit=True,
        bnb_4bit_quant_type="nf4",
bnb_4bit_compute_dtype="bfloat16"),
)

ds = load_dataset("json", data_files=
{"train":"train.jsonl","test":"dev.jsonl"})

trainer = SFTTrainer(
    model=model, tokenizer=tok,
    train_dataset=ds["train"], eval_dataset=ds["test"],
    peft_config=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"]),
    args=SFTConfig(
        output_dir="out", max_seq_length=1024, packing=True,
        per_device_train_batch_size=1, gradient_accumula-
tion_steps=16,
        gradient_checkpointing=True, num_train_epochs=2,
        learning_rate=2e-4, lr_scheduler_type="cosine",
warmup_ratio=0.03,
        logging_steps=10, eval_strategy="steps", eval_steps=50,
        save_steps=50, save_total_limit=2, bf16=True, seed=42,
    ),
)
trainer.train()
trainer.model.save_pretrained("out/adapter")
```

On 500 examples of about 600 tokens each, two epochs, this takes 30 to 60 minutes on a free notebook GPU.

The four hyperparameters that matter

Learning rate. For LoRA, `1e-4` to `2e-4`. Full fine-tuning wants roughly ten times lower, around `1e-5` to `2e-5`, and using a LoRA learning rate for a full tune destroys the model. If the loss goes to zero in fifty steps or turns to `nan`, the rate is too high.

Epochs. One to three on a few hundred examples. Small datasets overfit quickly, and the third epoch is often where the model stops generalising and starts reciting.

Effective batch size. Batch size times accumulation steps. Sixteen to thirty-two is a reasonable range. Too small makes training noisy; too large wastes the small dataset.

Max sequence length. Set it to your real data with `packing=True`, which fills each sequence with several examples rather than padding. On short examples packing can double throughput.

Everything else can stay at defaults for a first run.

Reading the curves honestly

Training loss falling means the model is fitting your examples. It says nothing about whether it does your task better.

Evaluation loss is more useful, and the shape to watch for is a fall then a rise: the point where it turns upward is where overfitting begins, and the checkpoint before it is the one you want.

Neither number is your metric. Loss is average token surprise. Your task is accuracy, or valid JSON rate, or whatever the product needs. A model can have lower loss and be worse for you — for instance because it learned to reproduce a verbose style that raises token-level agreement while making the extraction harder to parse.

Always run the actual task evaluation on the held-out set, at temperature 0, against the base model with your best prompt. Two numbers, same set, same settings. That comparison is the result; the loss curve is diagnostics.

What a failed run looks like

- **Loss barely moves.** Wrong target modules, learning rate far too low, or the loss is being computed on the wrong tokens. Print the trainable parameter count first.
- **Loss collapses to near zero in a few steps.** Learning rate too high, or the dataset has near-duplicates being memorised.
- **The model outputs the training format for every input, including irrelevant ones.** Overfitting. Fewer epochs, lower rank, more diverse data.
- **Output is fluent nonsense.** The chat template used in training does not match the one used at inference. Check both.

That last one is the most common of all, and the fix is to generate with the exact same template the trainer applied.

BEFORE YOU MOVE ON

After fine-tuning, evaluation loss is clearly lower than the base model's but task accuracy on the held-out set is slightly worse. What is the most reasonable interpretation?

A. The evaluation set is too small, so the accuracy difference is noise and the loss should be trusted.

B. The learning rate was too high, which lowers loss while damaging the model.

C. Loss measures token-level agreement with your training style, which the model learned, while accuracy measures the task, so the two can move in opposite directions.

D. The model has overfitted, and evaluation loss would rise if trained further.

56 · 9 min

Teaching it which answer is better

THE ONE THING TO KEEP

Preference tuning learns from pairs or binary judgements about which output is better, fixing verbosity, tone and refusal calibration that supervised tuning cannot — at an order of magnitude lower learning rate and with a strong tendency to make models longer.

The gap supervised tuning leaves

Supervised fine-tuning shows the model one good answer per input. It never sees a bad one, so it learns what a good answer looks like without learning what to avoid.

That leaves a specific set of problems unsolved: the model is too verbose, or too terse; it hedges when it should commit; it adds a summary paragraph nobody wanted; it refuses things it should handle and handles things it should refuse; it picks the second-best of two acceptable formats. These are preferences between outputs, and you often cannot write down the rule — you only know it when you see two answers side by side.

Preference tuning learns from exactly that: pairs of answers with a judgement about which is better.

The three methods you will meet

DPO — direct preference optimisation. Takes triples of prompt, chosen answer and rejected answer, and adjusts the model to raise the relative likelihood of the chosen one. It needs a frozen reference copy of the starting model, and a parameter `beta` controls how far the tuned model is allowed to drift from it. Lower beta means more change and more risk.

ORPO. Combines supervised and preference learning in a single stage with no reference model, so it is cheaper in memory and simpler to run. Useful when you are starting from a base model and would otherwise do two runs.

KTO. Needs only a binary judgement — this output was good, that one was bad — rather than matched pairs. This matters enormously in practice, because binary feedback is what real products collect. A thumbs-up button produces KTO data; producing DPO pairs requires someone to compare two answers to the same input deliberately.

```
from trl import DPOTrainer, DPOConfig
trainer = DPOTrainer(
    model=sft_model, ref_model=None,          # None uses the
adapter-disabled base
    args=DPOConfig(beta=0.1, learning_rate=5e-6, num_train_e-
pochs=1,
                   per_device_train_batch_size=1, gradient_ac-
cumulation_steps=16),
    train_dataset=pairs,                       # prompt / cho-
sen / rejected
    tokenizer=tok, peft_config=lora_cfg,
)
```

Note the learning rate: preference tuning wants roughly an order of magnitude lower than supervised tuning, around `5e-6` to `5e-7`. Using an SFT learning rate here is the standard way to destroy a model in twenty minutes.

Where the pairs come from

From your own model. Sample two completions at temperature 0.8 for each of 500 real prompts, and have a person pick the better one. Fast — a judgement takes seconds — and the pairs are on-distribution, which matters: preference data drawn from a different model teaches your model less.

From corrections. The original output is the rejected answer; the human-edited version is the chosen one. This is free if you are already correcting output, and it is the highest-quality source available.

From a judge model, with a rubric, spot-checked by a person. Cheap and biased in ways the judge is biased, so audit twenty judgements before trusting the rest.

Five hundred to two thousand pairs is the usual range. As with supervised data, consistency matters more than volume; pairs judged by three people with different tastes teach the average of three tastes, which is nobody's.

The order

Supervised tuning first, preference tuning second. Preference tuning adjusts a model that already does the task; it is a poor way to teach the task itself. Running DPO on a base model that has never seen your format mostly produces confusion.

The damage to watch for

Preference tuning is the stage most likely to make a model worse in ways your task metric will not show.

- **Verbosity drift.** Human raters, and judge models even more, prefer longer answers. Uncontrolled preference tuning reliably makes models more verbose. If your data was rated by a model, check length distributions before and after.
- **Capability loss.** Too many epochs or too low a beta pulls the model far from the reference and general ability degrades. One epoch is usually correct.
- **Reward hacking on format.** The model learns the surface features that correlated with being chosen — headings, bullet points, a confident opening — rather than the substance.

The defence is the same in every case: a regression set covering general capability, other languages and refusal behaviour, run before and after. The next lesson is about building one, and preference tuning is the stage that most needs it.

BEFORE YOU MOVE ON

After DPO on 1,500 pairs judged by a strong model, task accuracy is unchanged but responses are 60% longer and users complain. What most likely happened?

A. Beta was set too high, so the model drifted too far from the reference.

B. The pairs contained too few examples of short answers, so the model never learned brevity was acceptable.

C. Judge models systematically prefer longer answers, so the preference signal encoded length as a proxy for quality.

D. The learning rate was too low, so only superficial features such as length were learned.

57 · 9 min

Distilling a larger model into a smaller one

THE ONE THING TO KEEP

Response distillation is supervised fine-tuning on a teacher's outputs, so the student inherits the teacher's errors and its style, cannot exceed it, and must be measured against ground truth rather than against agreement with the teacher.

The idea, and the two versions of it

You have a large model that does the task well and is too expensive to serve. Distillation trains a small model to imitate it.

Response distillation is what almost everybody does. Run a set of prompts through the teacher, keep the outputs, fine-tune the student on those pairs. It is ordinary supervised fine-tuning where the labels came from a model instead of a person.

Logit distillation is stronger and rarer. Instead of the teacher's chosen token, you train against the teacher's full probability distribution at each step, minimising the divergence between student and teacher. The student learns not just what the teacher said but how uncertain it was, which carries much more information per example. It requires the same tokenizer for both models and access to the teacher's logits, which rules out any closed API and most convenient setups.

For most people, response distillation with a good filter on the outputs is the practical choice.

The legal part, which is not optional

Closed commercial APIs. Terms of service typically prohibit using outputs to train competing models. A great many popular community fine-tunes did exactly this, which is one reason their provenance sections are empty. Read the terms of the specific service; do not assume, and do not assume that because others did it, it is fine.

Open-weights teachers. Generally permitted, subject to the base licence. Apache 2.0 and MIT teachers impose nothing. Llama's terms permit training on outputs while requiring the resulting model's name to begin with Llama, plus the attribution obligations. Some licences restrict it outright. This is a five-minute check with a real answer, unlike most of the copyright material in this course.

Your own outputs. If the teacher is a model you are entitled to use commercially, distilling it is a straightforwardly clean route, and it is why open-weight teachers are worth the small quality gap.

A recipe that works

1. **Collect 5,000 to 50,000 real prompts.** From logs, ideally. Coverage matters more than count.
2. **Generate with the teacher at low temperature**, with the prompt you would actually ship.

3. **Filter.** This step is what separates a good distillation from a mediocre one. Drop outputs that fail schema validation, that are empty or truncated, that are refusals when a refusal was wrong, or that a cheap check finds inconsistent with the input. Expect to discard 10 to 30%.
4. **Have a person read 100 of them.** You are about to train on this; if 15% are wrong, your student learns to be wrong 15% of the time with total confidence.
5. **Fine-tune the student** with the previous lessons' settings.
6. **Evaluate against ground truth, not against the teacher.** Agreement with the teacher measures imitation. Only your labelled held-out set measures whether the student is any good.

The ceiling, and the inheritance

The student will not exceed the teacher on the distilled task, and normally lands somewhere below it. What it gains is cost: a 3B student serving at a fraction of a 70B's price, running where the teacher cannot.

It also inherits the teacher's **errors** and its **style**, and the second one is worth watching. Distilled models pick up the teacher's structure — the headings, the hedging, the closing summary — which makes them read as more capable than they are. Somebody evaluating by skimming five outputs will overrate a distilled model, which is exactly the failure the earlier vetting lesson described from the other side.

Reasoning distillation, which is a special case

Training a small model on a reasoning model's chains of thought produces a student that reasons in the same style. The well-known distills of DeepSeek-R1 into Qwen and Llama bases are the public examples, and they are genuinely better at multi-step problems than the bases they came from.

Two honest caveats. The gain is largest on problems resembling the distilled traces, mostly mathematics and code. And the student produces long chains, so it inherits the teacher's token cost per answer even though it is a smaller model — which removes part of the reason you wanted a smaller model.

BEFORE YOU MOVE ON

A distilled 3B student agrees with its 70B teacher on 94% of held-out prompts. What does that number establish?

A. The student is 94% as capable as the teacher, so the remaining gap is small.

B. It measures imitation only; if the teacher was wrong on 12% of these prompts, the student is confidently wrong on most of those too, and only a labelled set reveals it.

C. 94% agreement indicates the distillation dataset was too small, since a larger one would push agreement higher.

D. The 6% disagreement represents the student's improvements over the teacher.

58 · 8 min

Measuring what you broke

THE ONE THING TO KEEP

Fine-tuning moves the weights that held everything else, so report the task gain alongside a fifty-item regression set covering general instructions, other languages, alternative formats and refusals — and mixing general data into training usually keeps most of the gain with a fraction of the loss.

What happens to everything else

A fine-tune moves weights towards your examples. Those same weights encoded everything the model could already do. Move them far enough and previous capabilities degrade — this is catastrophic forgetting, and it is the reason a tuned model that scores brilliantly on your task can be quietly worse at half a dozen things you never measured.

The characteristic pattern after a small-dataset tune:

- The task improves, sometimes a lot.

- General instruction following weakens; the model answers in your trained format even when asked for something else.
- Other languages degrade, often noticeably, because they were never in your data.
- Refusal behaviour shifts in whichever direction your examples pushed it, including towards refusing nothing.
- Formatting outside your trained format gets worse.
- Long-context behaviour degrades if you trained on short examples only.

LoRA reduces all of this compared with full fine-tuning, because far fewer parameters move. It does not eliminate it, and higher ranks forget more.

The regression set

The defence is one artefact, built once, and it takes about an hour.

Fifty items, covering what the model must not lose:

- **10 general instruction-following items.** Summarise this, rewrite in a different tone, answer a general question, follow a two-part instruction.
- **10 items in each other language you serve.** Even if the tune is English-only, especially if the tune is English-only.
- **10 items with a different output format** from your trained one. If you trained JSON, ask for prose.
- **10 safety and refusal items.** Things that should be refused, and things near the line that should not be.
- **10 items from the original task at longer lengths** than your training examples.

Run it against the base model first and record the answers. That is your reference. Run it after every tune and compare. Score by reading — fifty items takes twenty minutes — or with a rubric and a judge model, spot-checked.

Report two numbers with every fine-tune: the task gain, and the regression change. A tune that gains 9 points on the task and loses 15 on the regression set is not an improvement, and without the second number it looks like one.

Reducing the damage

In order of effectiveness:

Fewer epochs. Most forgetting arrives in later epochs, after the task has already been learned. Two epochs instead of five is usually free.

Lower rank. Rank 8 forgets less than rank 64 and is enough for format and style tasks.

Mix in general data. Add 10 to 30% general instruction-following examples to your training set, drawn from an open instruction dataset. This is the standard mitigation and it works well. The mixture keeps the model anchored while it learns your task.

Lower learning rate, or fewer steps. Both reduce total movement.

Adapter rather than merge. An unmerged adapter can be turned off. A model serving both the tuned and untuned behaviour, selecting per request, sidesteps the trade entirely — the shipping lesson covers how.

The measurement that decides

Run three configurations on both sets: base model with your best prompt, tuned model, and tuned model with the mixed general data. Nine numbers, one afternoon.

Two configurations, measured on both sets

	Task score	Regression set
on task data only	**+9 points** the best headline number	**–15 points** other languages and formats degrade
eral data mixed in	**+8 points** almost all of the gain	**–2 points** a fraction of the loss

A tune that gains nine points on the task and loses fifteen on the regression set is not an improvement, and without the second number it looks like one. Run the base model with your best prompt as the third configuration, and ship the mixed run.

The result is usually that the mixed run gives almost all of the task gain with a small fraction of the regression, and that is the configuration to ship. Teams that skip this comparison ship the pure-task tune because it has the best headline number, and then spend a month diagnosing complaints that have nothing to do with the task they measured.

> *The honest framing: fine-tuning is not free capability, it is a trade. The regression set is how you find out what you traded, and it is the difference between a considered decision and an accident.*

BEFORE YOU MOVE ON

A fine-tune improves ticket-classification accuracy from 71% to 84% and the team ships it. Weeks later, support reports the assistant no longer handles Hindi messages well, though nothing about Hindi changed. What happened?

A. The tokenizer changed during fine-tuning, so Hindi is now segmented differently.

B. The English-only training data moved weights that also supported Hindi, degrading it, and no regression set was in place to catch it before release.

C. The adapter is being applied to Hindi requests when it should be disabled for them.

D. Quantisation applied at deployment damaged Hindi more than English.

59 · 8 min

Merging models, and what it really does

THE ONE THING TO KEEP

Merging averages fine-tuning deltas from a shared base, which works because those deltas stay in a connected low-loss region — and it is most reliable when merging your own runs and least trustworthy when a merge was selected on the benchmark it is reported against.

Averaging weights, and why it is not absurd

You can take two fine-tunes of the same base model and average their weights. The result often works, and sometimes performs better than either parent.

That sounds like it should produce nonsense, and the reason it does not is worth knowing: fine-tunes of a shared base stay in a connected low-loss region of the weight space. They have not travelled far from the base or from each other, so points between them are also reasonable models. Merge two models with *different* bases and you do get nonsense, immediately.

The hard requirements: identical architecture, identical tokenizer, same shapes. In practice, a shared base model.

The methods

Linear averaging and **SLERP** interpolate directly between two sets of weights, SLERP along a spherical path rather than a straight line. Simple and effective for two models.

Task arithmetic is the idea that clarifies the rest. Define a task vector as the difference between a fine-tune and its base — the change tuning made. Task vectors can be added to combine capabilities, or subtracted to remove one.

This gives a vocabulary for what merging is: you are doing arithmetic on the deltas, not on the models.

TIES merges many models by trimming small changes, resolving sign conflicts by a weighted vote, and averaging only the parameters that agree. It handles interference between task vectors, which plain averaging does not.

DARE randomly drops most of the delta and rescales what remains, on the observation that fine-tuning deltas are highly redundant. Often combined with TIES.

`mergekit` implements all of these from a short configuration:

```
models:
  - model: ./my-support-tune
    parameters: {weight: 0.5, density: 0.6}
  - model: ./my-multilingual-tune
    parameters: {weight: 0.5, density: 0.6}
merge_method: ties
base_model: Qwen/Qwen3-8B
dtype: bfloat16
```

Merging is cheap. It runs on a CPU with enough RAM, takes minutes, and needs no training data.

The use that reliably works

Merging your own runs. Train the same configuration with three seeds, or take several checkpoints from one run, and average them. This is the model-soup idea and it is the most dependable form of merging: it reduces variance, usually gains a little, and costs nothing but disk.

Combining your own domain adapters, where each was trained on a distinct capability you want together. Test the combination on both tasks; interference is common and TIES exists because of it.

The use that should make you cautious

Public leaderboards have been full of merges of merges, tuned by trying combinations until the score went up. That process optimises the benchmark directly, and the resulting models frequently underperform their headline numbers on anything else.

There are three further problems with a deep merge chain, and they are the reasons the vetting lesson treated them as a red flag:

- **Licence.** The merged model carries the obligations of every parent. A single non-commercial ancestor makes the whole thing non-commercial.
- **Provenance.** Nobody can say what data influenced the result, so no claim about the model's behaviour is supportable.
- **Evaluation.** Merges are frequently selected on the benchmark they are then reported against, which is the definition of overfitting.

How to merge responsibly

1. Merge models whose parents you know and whose licences you have checked.
2. Evaluate on a held-out set the merge selection did not use.
3. Include the regression set from the previous lesson; merges shift refusal behaviour and language coverage in ways nobody intends.
4. Record the recipe — the exact configuration, the parent revisions, the method — so the result is reproducible.
5. Prefer merging your own runs over merging strangers' models.

> *Merging is a genuinely useful tool with an unusually low cost and an unusually high rate of being used badly. The discipline that separates the two is entirely in step 2.*

BEFORE YOU MOVE ON

A team merges two Apache 2.0 fine-tunes with a third model that turns out to be a Llama derivative. What is the licence position of the result?

A. Apache 2.0, since two of the three parents are permissive and majority terms apply.
B. Unlicensed, since a merge of differently licensed models cannot be distributed at all.
C. The merge inherits every parent's obligations, so Meta's community licence terms, including the naming requirement, apply to the whole result.
D. No licence applies because merging produces new weights that are not derived from any single parent.

60 · 8 min

When the model does not have your language at all

THE ONE THING TO KEEP

Continued pretraining on raw text is the only fix for a language or notation the model never learned, it needs billions of tokens and vocabulary extension with mean-initialised embeddings, and it destroys instruction following unless followed by a fresh supervised tuning stage.

A different scale of intervention

Everything so far assumed the model already had the underlying capability and needed its behaviour shaped. Continued pretraining is for when it does not: a language it barely saw, a script it cannot tokenise, a domain with notation it has never encountered.

This is training, not tuning. It uses **raw text**, not instruction pairs, with the ordinary next-token objective, and it needs volume:

- **A domain** — legal filings, clinical notes, a company's technical corpus: hundreds of millions of tokens is a meaningful run.
- **A language the model saw a little of:** billions of tokens.
- **A language the model has essentially never seen:** tens of billions, and you are approaching the cost of training a small model from scratch.

For scale, an 8B model processing 10 billion tokens is in the region of a thousand GPU-hours on a datacentre card. That is a real budget, a real timeline and a real risk of a failed run, and it is why this is the least commonly justified item in this block.

Vocabulary extension

If the target language tokenises badly — the fertility measurement from an earlier lesson showing five or more tokens per word — you may need to add tokens before training.

The procedure: train a tokenizer on your corpus, take the merges the base tokenizer lacks, extend the vocabulary, and resize the embedding and output matrices. The new rows need sensible initialisation, and the technique that works is to initialise each new token's embedding as the mean of the embeddings of the pieces it used to be split into. Random initialisation makes the early training unstable and wastes a large amount of compute recovering.

Then train. Untrained new embeddings are worse than useless until they have seen enough text.

The step everybody forgets

Continued pretraining on raw text destroys instruction following. The model becomes a document continuer again, because that is what you just trained it to be. Ask the result a question and it produces more questions.

So the pipeline is three stages, and skipping the third produces a model people report as broken:

1. Continued pretraining on raw text in the target language or domain.

2. Supervised fine-tuning on instruction data — ideally in the same language, which usually means translating or writing an instruction set.
3. Optionally, preference tuning.

Stage 2 needs instruction data in the target language, and for underserved languages that data may not exist. Building it is often the larger part of the whole project.

The honest recommendation

For almost every team, the answer is not to do this.

The alternatives are better nearly always:

- **Use a model that already has the language.** The regional families from the earlier lesson exist precisely because organisations with the right corpora and the right budget already did this work.
- **Fine-tune for register and format** on a few hundred examples, which fixes what usually looks like a language problem and is a day of work.
- **Retrieve in the target language**, which fixes what is actually a knowledge problem.
- **Use a translation stage**, accepting its costs, if the output is machine-consumed rather than read.

Continued pretraining is justified when you have a genuinely underserved language, a corpus in the billions of tokens, the compute, and a reason nobody else will do it — which describes a research group, a national initiative or a well-funded regional effort, and very few product teams.

> *If you are in that position, the good news is that the recipe is public: AI2's OLMo releases their data, code and checkpoints, and several regional projects have published their continued-pretraining pipelines in enough detail to follow. That is the practical value of the fully-open end of the openness spectrum, and this is the case where it matters most.*

BEFORE YOU MOVE ON

A team completes continued pretraining on 4 billion tokens of a low-resource language. The model now writes that language fluently but responds to every question with more text on the same topic rather than an answer. What is missing?

A. The chat template was not applied during pretraining, so the model cannot recognise conversation structure.
B. Vocabulary extension was incomplete, so instruction tokens are being mis-tokenised.
C. A supervised instruction-tuning stage in the target language, since raw-text training returns the model to document continuation.
D. The learning rate was too high, which erased the model's alignment.

61 · 8 min

Shipping the adapter

THE ONE THING TO KEEP

An unmerged adapter lets one base model serve many tuned behaviours and lets untuned traffic bypass the tune entirely, and a fine-tuned deployment must pin the base revision, adapter, prompt, sampler and both evaluation numbers as a single versioned unit.

Merge, or serve the adapter

Merged. Fold the adapter into the base and save full weights. The result loads anywhere, converts to GGUF, and needs no special support. It costs the full model size on disk and per deployment, and you cannot turn it off.

Unmerged. Keep the adapter as a separate 40 to 200 MB file. Serving stacks that support it can load one base model and many adapters, choosing per request.

That last capability is the one worth designing around. vLLM will hold a base model in memory and route each request to whichever adapter it names:

```
vllm serve Qwen/Qwen3-8B   --enable-lora   --max-loras 8 --max-
lora-rank 32   --lora-modules support=/adapters/support-v3
triage=/adapters/triage-v1
```

```
{"model": "support", "messages": [...]}
```

One GPU, one copy of the base weights, eight tuned behaviours. For a multi-tenant product, or a team with several tasks, this is the difference between one machine and eight. The per-request cost of applying an adapter is small.

It also gives you the fix from the forgetting lesson: requests that need general behaviour can address the base model directly, so a tune that damaged general capability damages only the traffic that wanted the tune.

For laptop deployment you will merge and convert:

```
python convert_hf_to_gguf.py ./merged --outfile ft-f16.gguf --
outtype f16
./llama-quantize ft-f16.gguf ft-Q4_K_M.gguf Q4_K_M
```

Verify the chat template survived the conversion, for the reasons the conversion lesson gave. This is the most common way a good fine-tune arrives broken.

Version everything together

A fine-tuned deployment has five moving parts, and all five must be pinned as one unit:

```
base model    Qwen/Qwen3-8B @ 9a3f21c
adapter       support-v3 (sha256 …)
prompt        prompts/support.qwen3-8b.md @ commit e41b0
sampler       temperature 0, top_p 1, repeat_penalty 1.0, max_-
tokens 400
eval          held-out set v4, task 0.86 / regression 0.94
```

Changing any one of them changes the behaviour. A team that upgrades the base model without re-tuning, or edits the prompt without re-running the evaluation, has shipped an untested system that looks like the tested one.

Keep this block in the repository next to the code, not in somebody's notes.

The gate

Before any adapter is promoted, two numbers on two fixed sets:

- **Task metric on the held-out set**, compared against the base model with the best prompt, at identical settings.
- **Regression score**, compared against the recorded base answers.

Write both into the release. If the task gain does not exceed the noise on a set that size, do not ship it — an earlier lesson gave the arithmetic, and on forty examples a five-point gap is nothing.

Automate this as a script that takes an adapter path and prints two numbers. It takes an afternoon to write and it removes the entire class of argument about whether the new tune is better.

Rollback, and the copy that saves you

Keep the previous adapter and the ability to switch by configuration. Adapters are small enough that keeping every version you ever shipped costs almost nothing, and being able to revert in one deploy rather than one retrain is worth far more than the disk.

Keep the base model weights too, at the exact revision, on your own storage. The lesson on why labs release weights explained why: your fine-tune is

meaningless without the base it was trained against, and a repository can be renamed, gated or withdrawn. An adapter without its base is an orphan.

> *The whole of this lesson reduces to one sentence: a fine-tune is a base revision plus an adapter plus a prompt plus a sampler plus two measurements, and shipping any of them separately is how a working system becomes a mystery.*

BEFORE YOU MOVE ON

A team merges their adapter into the base, deploys the merged weights, and later finds that general requests are handled worse than before. What capability did merging remove?

A. The ability to quantise the model, since merged weights cannot be converted to GGUF.

B. The ability to route general traffic to the unmodified base while tuned traffic uses the adapter.

C. The ability to measure the regression, since the base is no longer available for comparison.

D. The ability to update the adapter without retraining, since merging fixes the weights permanently.

CASE STUDY · MODULE 6

Nine points better, and worse at everything else

A five-person company in Nairobi selling practice-management software to forty veterinary clinics, fine-tuning a small model to turn a vet's dictated visit note into a structured record.

The product worked like this: a vet finished a consultation, spoke a note into the tablet, and the software produced a structured visit record — species, presenting complaint, findings, treatment given, follow-up date — which the vet checked and saved. The structuring was done by a 4B model with a carefully written prompt, eight examples in the context, and a JSON schema.

It was good and it was expensive to run in a way that mattered. The eight examples came to nine hundred tokens on every request, and with forty clinics and roughly sixty consultations a day each, that was a large and entirely repetitive bill. Wanjiru, who had built it, made the standard argument for fine-tuning: put the examples into the weights, drop the prompt to three sentences, and the per-request cost falls by most of a factor of four.

She did it properly. Six hundred real notes pulled from consenting clinics, each structured record corrected by a veterinary nurse who had done the job for years. A held-out set of a hundred she touched once. A LoRA at rank sixteen on all the linear layers, two epochs, an hour on a free notebook GPU.

The result was a tuned model that scored **eighty-seven per cent** on the held-out set against **seventy-eight** for the base model with her best prompt. Nine points, measured at identical settings, on real notes. The prompt shrank from nine hundred tokens to forty.

The second measurement

Wanjiru had built a fifty-item regression set at the start of the project, because a colleague had insisted, and she had not expected to need it. Ten general instruction-following items, ten in Swahili, ten asking for output in a format

other than the trained one, ten refusal items, and ten longer notes than anything in training.

The tuned model lost fifteen points on it.

The damage was specific and, once seen, obvious. Swahili degraded most — there was no Swahili in the six hundred training notes, because the clinics that had consented to share records were the ones that wrote in English. Several clinics in the west of the country dictated in a mix of Swahili and English and the tuned model handled them visibly worse than the base had. Worse still, it had learned that the answer to anything is a JSON object: asked to write a plain-language discharge summary for a client, a feature two clinics used daily, it produced a visit record.

The deadline

A national rabies-vaccination campaign was eleven days away. Campaign weeks triple consultation volume, which was precisely when the token saving was worth most and precisely when the software could least afford to be strange.

Ship the tuned model. Nine points better on the thing they measured in the pitch deck, a four-fold cut in prompt cost during the busiest fortnight of the year, and a known, unmeasured degradation in Swahili and in the discharge-summary feature — for clinics that had not been asked and would not know why.

Ship the base model with the long prompt. Nothing breaks, nothing improves, and the bill through the campaign is what it always was.

Re-run with general data mixed in. Add twenty per cent general instruction-following examples and a hundred Swahili notes gathered from two clinics that would give them, then re-measure both sets. Wanjiru estimated four days including the labelling, which left a week of margin and assumed nothing went wrong.

Her co-founder's position was that a nine-point gain is the number a customer understands and the fifteen-point loss is a number no customer will ever ask about. That is true, and it is exactly why the second number has to be reported

alongside the first: the only person who will ever measure it is the team that made the tune.

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-ran with the mixture and it took five days rather than four. The mixed tune scored eighty-six per cent on the task — one point below the pure tune, comfortably inside the noise of a hundred-item set — and lost two points on the regression set instead of fifteen. Swahili came back to within a point of the base model and the discharge-summary feature worked again. They shipped it four days before the campaign. Two practices outlived the deadline: the regression set is now run against every adapter before promotion, by a script that takes an adapter path and prints two numbers, and the deployment pins the base revision, the adapter hash, the prompt commit, the sampler settings and both measurements as a single versioned block in the repository. When the base model was superseded seven months later, that block was what made the re-tune a day's work instead of an archaeology project.

The pure tune gained nine points on the task and lost fifteen on the regression set. Why is the second number not simply a smaller version of the first?

Because they measure different populations. The task number covers the inputs the tune was built for, which are the inputs the team looks at. The regression number covers everything the model could already do — other languages, other output formats, general instructions, refusals — which is where the weights that moved were previously doing their work. A customer will report the task improvement and will never file a bug saying the Swahili got worse; they will simply use the feature less. The only people who can detect that are the ones who measured it before shipping.

Mixing twenty per cent general data cost one point of task score and recovered thirteen points of regression. Why does that mixture work?

Fine-tuning moves the weights towards your examples, and general examples in the training mix keep pulling them back towards the behaviour the base already had, so the model learns the task without drifting as far from everything else. It is the standard mitigation because the trade is usually this lopsided: almost all of the gain for a fraction of the loss. Fewer epochs and a lower rank do the same job more bluntly, by reducing total movement, and are worth trying first when the mixture is not available.

Wanjiru's Swahili degradation traced back to a property of her dataset, not of the tuning method. Which property, and what does it imply about how training sets are collected?

Diversity of inputs. The six hundred notes came from the clinics that consented, and those clinics wrote in English, so a register used daily by other customers was entirely absent from training. A model inherits every property of its dataset, including the shape of who happened to be willing to share. Collection has to be sampled deliberately across time, source and user type rather than taken from whatever was easy to obtain, and the absence of a category in the data is a prediction about where the model will get worse.

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 team wants a model to know four thousand internal product facts and plans a LoRA on documents describing them. What should they expect?

A. Correct facts but a degraded output format, which retrieval cannot repair

B. The same result as retrieval at a lower serving cost, because no context is needed

C. Reliable recall of the facts, since the weights now contain them

D. Unreliable recall, contradiction under rephrasing, and losses elsewhere

2 A team raises LoRA rank from 16 to 64 and leaves alpha at 32. The run completes and the model barely differs from the base. Why?

A. The B matrix starts at zero, and at rank 64 it takes far longer to leave zero

B. Rank 64 needs more epochs, and two was not enough to move the weights at all

C. The update is scaled by alpha over rank, so raising rank weakened it

D. Higher ranks always converge more slowly because there are more parameters

3 A QLoRA run on a sixteen-gigabyte card fails with an out-of-memory error. Which change should you make first?

A. Reduce the LoRA rank, since the adapter is what the run is training

B. Turn on gradient checkpointing and cut the sequence length

C. Switch the optimiser from Adam to plain stochastic gradient descent

D. Quantise the base further, from four bits down to three, to free weight memory

4 Two careful annotators independently label thirty of your examples and agree on twenty-four of them. What does that tell you before any training starts?

A. Roughly eighty per cent is the ceiling your model can be measured against

B. The task is unsuitable for fine-tuning and the effort should go to retrieval instead

C. One annotator is wrong, and their disputed items should be dropped from the set

D. The dataset needs about six hundred more examples to overcome the disagreement

5 A fine-tune gains nine points on the task metric. Which single artefact tells you whether that is an improvement?

A. The training loss, compared against the previous run at the same rank

B. A public benchmark such as MMLU, run on both the base and the tuned model

C. The evaluation loss curve at the final saved checkpoint

D. A fifty-item regression set run before and after

6 Averaging the weights of two fine-tunes of one base often works. Averaging two models with different bases gives nonsense immediately. What explains the difference?

A. Merging requires identical random seeds, which only a shared training run provides

B. Averaging only works when both models were trained on the same dataset

C. Tunes of one base stay close together in a low-loss region

D. Different bases were quantised differently, so their weights sit on different scales

MODULE 7

Judging models and claims

Every part of this ecosystem asks you to believe a number: a benchmark table, a leaderboard position, a quantisation that is claimed to be lossless, an embedding model at the top of a chart, a price per million tokens. This block is about generating your own evidence instead — reading a model card for what it does not say, running the standard harness yourself, proving what a quantisation cost, measuring latency and cost per task properly, and swapping a model under a live product without breaking it.

> **BY THE END OF THIS MODULE YOU CAN**
>
> Settle a model choice with evidence you produced yourself — a private held-out set, a pinned harness, a quantisation compared against its full-precision reference, latency measured at real concurrency and cost expressed per task rather than per token — and explain to a sceptical colleague which published claims about that model you believe and why

62 · 9 min

What Benchmarks Actually Measure

THE ONE THING TO KEEP

A benchmark result is evidence about that benchmark; transfer to your task is a hypothesis you have to test.

A score is not a property of a model

A benchmark number is produced by a pipeline: the model, plus the prompt format, plus the number of examples shown, plus the decoding settings, plus the regex that pulls an answer out of the text, plus the harness that glues it together. Change any one and the number moves, often by several points.

This is why two reports of "MMLU" for the same model can differ. Nobody lied. They ran different evaluations with the same name. When you see a comparison, the first question is not "how big is the gap" but "was this the same pipeline for both".

What actually produces a benchmark number

The model	The only part named when somebody quotes the score
The prompt format	Chat template applied, or raw. Several points on an instruct model
The shot count	Zero-shot and five-shot differ substantially on the same model
The decoding settings	Greedy or sampled, and at what temperature
The answer extraction	The expression that pulls a letter out of free text. It changes between harness versions
The harness, at a commit	Task definitions get fixed, so last year's score is not comparable with this year's

Change any layer and the number moves, often by several points, which is why two honest reports of the same benchmark for the same model disagree. The first question about a comparison is not how big the gap is, but whether both sides ran the same pipeline.

Contamination

Benchmark test sets live on the public web. Models are trained on the public web. Some of the test set ends up in training, and the model recites rather than solves.

The cleanest evidence is GSM1K, published in 2024: researchers built a fresh set of grade-school maths problems matched in style and difficulty to GSM8K, and re-tested. Some model families dropped by up to around thirteen points. Others were flat. The size of the drop is a decent estimate of how much of the original score was memory.

Most contamination is accidental. Nobody has to cheat for a widely mirrored test set to land in a web crawl. But accidental contamination inflates the number just as much as deliberate contamination does.

Gaming, in ascending order of dishonesty

1. **Picking the favourable configuration.** Trying several prompt formats and shot counts and reporting the best. Nearly universal, rarely disclosed.
2. **Undisclosed aggregation.** Reporting majority vote at k, or best-of-n, next to someone else's single-sample number. A maj@64 score is not a score you will ever see in production.
3. **Training on benchmark-shaped data.** Adding a large pile of grade-school word problems is a legitimate training choice that also destroys GSM8K's ability to predict anything else.
4. **Training on the test set.** Rare among reputable labs. Not unknown.

Arenas measure preference, which is a different thing

Human preference leaderboards are harder to contaminate, because the prompts are fresh. They measure what people pick when shown two answers, which rewards length, headers, bullet lists, confident tone, and agreement with the asker. Style-controlled variants of these rankings exist and are the ones worth reading. Note also that models are often tested under anonymous codenames, and only the results a lab likes get claimed.

The benchmark itself has a noise floor

An audit of a 3,000-question sample of MMLU found problems with roughly 6.5% of the items — ambiguous questions, wrong keys, unanswerable prompts. Cleaned and harder variants exist for that reason. When two models are within two or three points, a meaningful share of the difference is the benchmark's own errors. Hugging Face's Open LLM Leaderboard was eventually archived rather than patched, which tells you how far saturation and contamination had gone.

What still carries information

- **Held-out or refreshed sets.** Evaluations that rotate questions monthly, or keep a private split, cannot be trained on in advance.
- **Benchmarks with an execution harness.** Code that must compile and pass tests, or agent tasks that must actually complete. Something has to work, which is much harder to fake than a multiple-choice letter.
- **Papers that publish ablations and failures.** A lab showing where its model is worse is giving you calibration on the numbers where it is better.
- **Twenty outputs on your own inputs.** This beats every leaderboard for your decision, and the last lesson is about doing it.

Six questions for any launch post

Which harness. How many shots. Single sample or aggregated. Who ran the baseline numbers — them, or copied from another paper. Is the gap larger than three points. Does the benchmark look anything like your task.

If you cannot answer four of those, the post is marketing. That does not mean the model is bad. It means the post is not evidence.

BEFORE YOU MOVE ON

Model A's launch post shows it 2.1 points above Model B on MMLU, with B's figure copied from B's own paper. What is the most defensible read?

A. The comparison is uncontrolled and the gap sits inside the benchmark's own error and harness variation, so it supports almost nothing.

B. A is genuinely stronger on knowledge tasks, since across roughly fourteen thousand questions a 2.1 point gap is statistically significant.

C. The comparison is fair because MMLU is a standard benchmark, so any two reported MMLU scores are directly comparable.

D. Discount it completely, since self-reported benchmark numbers are fabricated as a rule.

63 · 8 min

Reading a model card for what it does not say

THE ONE THING TO KEEP

A model card's front matter is structured and checkable while its prose is a set of claims, and a benchmark number without the harness commit, shot count, aggregation, decoding settings and self-run baselines is not reproducible evidence.

Two documents in one file

A model card is part machine-readable metadata and part marketing prose, and the two deserve different levels of trust.

The **front matter** — the YAML block at the top — is structured, is used by tooling, and is the hardest part to make vague. `license`, `base_model`, `language`, `pipeline_tag`, `library_name`, `datasets`, `tags`. Read this first. It takes fifteen seconds and it settles the licence chain, the parent model, and what the publisher is claiming about language coverage.

The **prose** is where claims live. Read it with a specific question in mind: for each claim, what evidence would settle it, and is that evidence here?

The sections, and what each is worth

Model description. Architecture, size, context length, what it was built for. Mostly factual and mostly reliable.

Intended use, and out-of-scope use. The presence of a real out-of-scope section — specific, not boilerplate — is the strongest positive signal on the page. It means somebody thought about failure. A card whose out-of-scope section says only *illegal purposes* has told you nothing.

Training data. Almost always a sentence. *A mixture of publicly available data* means you know nothing, and the openness rubric from the first block scores it zero. Where a card names datasets, follow the links; some of them do not exist.

Evaluation. The section that most needs interrogating, next.

Limitations and bias. Frequently copied between cards. Specific limitations — *degrades above 16k tokens*, *weak at Bengali despite the language tag* — indicate real evaluation. Generic paragraphs about how models can produce biased output indicate a template.

Interrogating a benchmark table

A number in a model card is worth something only with five accompanying facts:

1. **Which harness, at which commit.** Without this the number is not reproducible even in principle.
2. **How many shots.** Zero-shot and five-shot MMLU differ by several points on the same model.
3. **Single sample or aggregated.** A majority-vote-at-64 score sitting in a column next to somebody else's single-sample number is not a comparison.
4. **The decoding settings.** Greedy or sampled, and at what temperature.

5. **Who produced the baseline numbers.** Numbers the publisher ran themselves, under the same configuration, are a comparison. Numbers copied from three different papers are a collage.

If four of the five are missing, the table is a claim, not evidence. That does not mean the model is bad; it means you have to measure it yourself, which you were going to do anyway.

Specific things to be suspicious of

A language list far longer than the evaluation. A card listing 100 languages and evaluating on four has told you which four it was tested on.

Only the benchmarks the model wins. Everybody reports MMLU. If a code model omits code benchmarks, ask why.

Comparison charts with no axis label, or with a truncated axis that turns two points into a visual chasm.

State of the art with no date. In this field the phrase decays in weeks.

No `base_model` on an obvious derivative. Either carelessness or a licence chain somebody preferred not to draw attention to.

Very high scores from a very small organisation with no accompanying method. Occasionally a real result. More often a contaminated evaluation, and always worth reproducing before quoting.

A three-minute procedure

```
hf download <org>/<model> README.md --local-dir ./card
sed -n '1,30p' ./card/README.md          # front matter: li-
cence, base, languages
grep -in "out-of-scope\|limitation\|not intended"
./card/README.md
grep -in "harness\|shots\|few-shot\|lm-eval\|temperature"
./card/README.md
```

The third command is the one that tells you the most. If it returns nothing, no number on the page is reproducible, and you now know exactly how much weight to put on the table.

What a good card looks like

For contrast, the shape worth emulating and worth trusting: front matter complete and consistent; a paragraph on intended use and a specific out-of-scope list; named datasets with links; an evaluation table with the harness commit, shot count and decoding settings, reporting the base model under the identical configuration; a limitations section naming real weaknesses; and a contact route.

Cards like that exist. They are a minority, they are disproportionately from the fully-open end of the spectrum, and the effort visible in them is itself evidence about the care taken with the model.

BEFORE YOU MOVE ON

A model card reports 79.4 on MMLU next to a competitor's 74.1, citing the competitor's figure from that model's own release post. What is the specific problem?

A. MMLU is contaminated, so no comparison using it means anything.

B. The two numbers came from different evaluation pipelines — different harness, shots, decoding and possibly aggregation — so the gap may be entirely procedural.

C. The competitor's number is out of date and a newer version would score higher.

D. A 5.3-point gap is within benchmark noise, so neither number is meaningful.

64 · 9 min

The leaderboards that still carry information

THE ONE THING TO KEEP

Leaderboards die from saturation and contamination, so the ones that still carry information use private or rotating items, execution-based verification or fresh human prompts — and even then their proper use is shortlisting, not deciding.

What kills a leaderboard

Two things, and both are predictable. **Saturation**: everybody scores above 85 and the remaining spread is the benchmark's own error rate. **Contamination**: the test set is public, so it is in the training data, and the ranking measures exposure rather than capability. Hugging Face's Open LLM Leaderboard was eventually archived rather than patched, which is the clearest statement anyone has made about how far both had gone.

So the question is not which leaderboard is best, but which design properties resist those two failures. There are four.

A private or rotating test set. Items that cannot be trained on because they were not published, or were published after the model was trained.

Execution-based verification. Code that must compile and pass tests, or a task that must actually complete. Something has to work, which is far harder to fake than picking a letter.

Fresh human prompts. People type new questions, so there is nothing to memorise — at the cost of measuring preference rather than correctness.

A published harness. If you can rerun it yourself on a model you understand, you can calibrate the numbers.

The ones worth reading, and what each measures

Pairwise human preference arenas. Two anonymous answers, a person picks one, ratings computed with a Bradley-Terry or Elo-style model. Resistant to contamination because the prompts are new. Measures *preference*, which rewards length, headers, confident tone and agreement with the asker — which is why the style-controlled variant of the ranking is the one to read. Note also that models are often tested under codenames, and only results a lab likes get claimed publicly.

Rotating-question benchmarks. Evaluations that add new items monthly and score models only on items published after their training cut-off. This directly targets contamination, and it is the most useful recent development in public evaluation.

Execution-based code and agent benchmarks. Tasks against real repositories where a patch must make failing tests pass. Expensive to run, hard to fake, and the closest public proxy for whether a model can do work.

Domain leaderboards — embeddings, retrieval, speech recognition, translation, long context. Usually narrower and often more honest, because the communities are smaller and the metrics are older and better understood. Each has its own contamination story worth reading before quoting.

Price and speed trackers. Independent measurement of tokens per second, time to first token and price across providers. These measure a real and stable thing, and they answer a different question from quality — which, given the earlier lesson on providers serving the same model differently, is a question you actually need answered.

Reading a ranking properly

Look for the interval. Preference ratings come with confidence intervals, and adjacent models routinely overlap. A 15-point gap with overlapping intervals is not a ranking, it is a tie displayed in an order.

Check the sample size behind each entry. A model with 800 votes and one with 40,000 are not measured to the same precision.

Check the date. A three-month-old leaderboard in this field describes a different landscape.

Check what was actually run. Which quantisation, which context length, which provider — the same warnings from the providers lesson apply to leaderboard entries served through an API.

What no leaderboard can do

None of them evaluates your task, your documents, your languages, your latency budget or your licence constraint. The correct use of a leaderboard is **shortlisting**: it narrows thirty candidates to three that are plausibly in the right class. The choice between those three is made on forty of your own examples, and that measurement outranks every public number.

> *The one-sentence test for whether a leaderboard deserves your attention: can you tell, from its documentation, exactly what was run against what, and could you reproduce one row of it? If not, it is entertainment.*

BEFORE YOU MOVE ON

Model A sits 12 rating points above model B on a human preference arena, with confidence intervals of plus or minus 9 points on each. What can you conclude?

A. Model A is better, since the point estimate is higher and intervals are conservative.

B. The intervals overlap substantially, so the ordering is not established, and in any case preference is not correctness.

C. Model A is better on style but worse on substance, which is what preference rankings measure.

D. More votes would resolve it, so the ranking is correct but imprecise.

65 · 9 min

Running the standard harness yourself

THE ONE THING TO KEEP

Running lm-evaluation-harness yourself against any OpenAI-compatible endpoint is how you calibrate published claims and catch regressions, and your number will differ from a paper's mostly through shot count, chat template and harness version — so pin the commit and log the samples.

Why run it at all

You have been told repeatedly in this course that public benchmarks are weak evidence for your task. They remain useful for two things: **calibration**, so you know whether the number a card reports is reproducible at all, and **regression checking**, so you know whether a quantisation, a fine-tune or a provider change broke something general.

The community standard is EleutherAI's `lm-evaluation-harness`, which is what most reported numbers come from and what you should therefore run.

The basic run

```
pip install lm-eval

lm_eval --model hf   --model_args pretrained=Qwen/Qwen3-8B,re-
vision=9a3f21c,dtype=bfloat16   --tasks
arc_challenge,hellaswag,gsm8k   --num_fewshot 5   --batch_size
auto   --output_path ./results   --log_samples
```

`--log_samples` writes every prompt and completion to disk. Turn it on always. The aggregate number tells you very little; reading twenty failures tells you what the model actually did.

For a quick check, `--limit 200` runs a subset. Useful for iteration, not for a number you report.

Evaluating anything with an endpoint

The important capability, and the one people miss: the harness can run against any OpenAI-compatible server. That means you can evaluate a GGUF served by llama.cpp, a quantised model in vLLM, a fine-tune, or a rented endpoint — all with the identical harness:

```
lm_eval --model local-chat-completions   --model_args
model=qwen3-
8b,base_url=http://localhost:8080/v1,num_concurrent=4   --tasks
gsm8k --apply_chat_template --num_fewshot 5
```

This is how you compare a quantisation against its reference, or a provider against another, without changing anything else in the pipeline.

`--apply_chat_template` matters. Benchmarks were historically run on base models with raw prompts. Running an instruct model without its template gives a lower number and running it with the template gives a different one; both appear in the wild. Whichever you choose, be consistent and say which you did.

Why your number differs from the paper's

Expect a difference, and expect it to be a few points. The causes, in rough order of size:

- **Shot count.** Five-shot and zero-shot differ substantially.
- **Chat template applied or not.** Several points on instruct models.
- **Answer extraction.** How the harness pulls the answer out of free text — a regular expression, a letter match, a normalisation step. Task implementations vary and change between harness versions.
- **Harness version.** Task definitions get fixed. A score from last year's version is not comparable with this year's.

- **Precision and quantisation.** Small, but real.
- **Batch size.** Non-determinism from batched floating-point reductions occasionally flips a near-tie.

Which is why the discipline is to **pin the harness commit** alongside the model revision, and to report both.

Your own task, in the same harness

The harness accepts custom tasks as YAML, so your private evaluation can live next to the public ones and be run with one command:

```
task: support_triage
dataset_path: json
dataset_kwargs:
  data_files: {test: ./data/triage_test.jsonl}
output_type: generate_until
doc_to_text: "Ticket: {{ticket}}\nLabel:"
doc_to_target: "{{label}}"
generation_kwargs: {until: ["\n"], temperature: 0.0}
metric_list:
  - metric: exact_match
```

This is worth the hour it takes. It gives you one command that produces both the public numbers and yours, at the same settings, logged the same way — which is what makes a comparison across models honest rather than assembled.

What to actually run

For a model swap or a quantisation change, a compact set is enough: one general knowledge task, one reasoning task, one instruction-following task, one in each language you serve, and your own task. Six numbers, half an hour on a small GPU, and it catches the large regressions.

Do not run forty tasks. The time cost is real and the extra numbers do not change decisions, because you already know that the decision belongs to your own held-out set.

BEFORE YOU MOVE ON

A team's MMLU score for an instruct model comes out four points below the published figure, using the same harness and the same shot count. What is the most likely cause?

A. Their download is a different quantisation from the one the publisher evaluated.

B. The harness is sampling rather than using greedy decoding, adding variance.

C. One run applied the model's chat template and the other did not, which shifts instruct-model scores by several points.

D. MMLU contamination affects the publisher's copy of the model but not theirs.

66 · 9 min

Proving what a quantisation cost

THE ONE THING TO KEEP

Perplexity is a screening test; the sharper measurement is divergence from the full-precision model on the same corpus, reported as the fraction of tokens where the top choice differs, plus agreement on your own examples at long context and in your languages.

Four methods, from weakest to strongest

You have a Q4 file and a claim that it is nearly lossless. Here is how to check, in ascending order of usefulness.

1. Perplexity

```
./llama-perplexity -m model-f16.gguf -f wiki.test.raw -c 4096
./llama-perplexity -m model-Q4_K_M.gguf -f wiki.test.raw -c
4096
```

Perplexity is the exponentiated average negative log-likelihood on a fixed corpus — how surprised the model is by text it did not write. Lower is better, and the comparison is only meaningful between two models sharing a tokenizer, which is exactly the quantisation case.

A rise from 6.10 to 6.18 is small. A rise to 7.4 is not. But two quantisations can sit within a hundredth of each other and behave differently on structured output, so perplexity is a screening test, not a verdict.

2. Divergence from the reference

Much more sensitive, and the right default. Instead of asking how surprised each model is by a corpus, ask **how far the quantised model's output distribution has moved from the full-precision model's**.

```
# produce a reference from the unquantised model
./llama-perplexity -m model-f16.gguf -f wiki.test.raw --kl-divergence-base ref.dat

# compare a quantisation against it
./llama-perplexity -m model-Q4_K_M.gguf -f wiki.test.raw --kl-divergence ref.dat
```

The output includes the mean Kullback-Leibler divergence and, more usefully, the **fraction of tokens where the top choice differs** between the two models. That last figure is directly interpretable: if the quantised model picks a different most-likely token 1.5% of the time, you know what you bought. Above about 5%, expect visible behavioural differences.

Report the tail as well as the mean. A quantisation with a low mean divergence and a heavy 99th percentile is one that is usually identical and occasionally very different, which is worse for a product than a uniform small degradation.

3. Your own task

Run your forty held-out examples through both files at temperature 0, and compare the answers directly. Two numbers matter: the task score for each,

and the **agreement rate** between them.

Agreement is the sharper instrument on small sets. With forty examples, a two-point score difference is noise; but if the two models produce different answers on nine of forty items, something real has changed even when the scores happen to match.

4. The specific capabilities quantisation damages

From the earlier lesson: long-context retrieval, lower-resource languages, multi-step arithmetic and long-output instruction adherence degrade before short English chat does. So test those deliberately rather than hoping a general set covers them.

- Ten questions whose answers sit deep in a long document, at your real context length.
- Ten items in each non-English language you serve.
- Ten items requiring a strict output format held over a long response.

This is thirty items and it is where the difference will appear if there is one.

Test the cache separately

KV cache quantisation is a different decision from weight quantisation and is frequently changed at the same time, which confounds both. Run four configurations if the setting matters to you: full weights with full cache, quantised weights with full cache, full weights with quantised cache, and both. The long-context items are where cache quantisation shows up, and it is often the cause when a model that tested well behaves oddly on long documents.

What to write down

For any quantisation you ship:

```
model      Qwen3-8B @ 9a3f21c
quant      Q4_K_M (imatrix, bartowski, rev 2f8c1)
perplexity 6.10 -> 6.17 on wiki.test.raw, c=4096
top-token  differs on 1.4% of tokens vs f16
task       0.86 -> 0.85 on held-out v4 (40 items), agreement
37/40
long-ctx   0.90 -> 0.80 on the 10 deep-retrieval items
decision   accepted; long-context path routed to Q6
```

Six lines, and it turns a choice somebody made once into a documented decision anybody can revisit.

BEFORE YOU MOVE ON

Two quantisations of the same model have almost identical perplexity, but one changes the model's top-choice token on 0.9% of positions and the other on 6%. What follows?

A. Perplexity is the more reliable metric, so the two quantisations are equivalent in practice.

B. The 6% quantisation will behave visibly differently in generation, and perplexity failed to separate them because it averages likelihood rather than tracking the chosen token.

C. The 6% figure indicates a corrupted file, since a valid quantisation cannot differ that often.

D. Top-token disagreement is expected to be high at any bit width because sampling is stochastic.

67 · 9 min

Evaluating an embedding model on your corpus

THE ONE THING TO KEEP

Retrieval is cheap to evaluate on your own corpus with 50 labelled queries, and the grid of embedding model against chunk size usually shows chunking moving recall more than the model does.

Why the leaderboard will not decide this

MTEB is the standard embedding benchmark and it has two problems that matter here. Models are routinely trained on the training splits of MTEB's own tasks, so the aggregate increasingly measures familiarity with those datasets. And the tasks are dominated by English and by web-shaped text; if your corpus is support tickets in Hindi, or contracts, or product listings, the ranking transfers weakly.

The good news is that evaluating retrieval on your own corpus is genuinely easy — much easier than evaluating a language model — because there is a right answer and it is cheap to check.

Building the set: two hours

1. **Take 50 real queries** from your logs. If you have no logs, write them by asking three colleagues what they would type, which is worse but workable.
2. **For each, find the document or chunk that answers it.** Mark it. Where several are relevant, mark them all. This is the only labour, and 50 queries is about ninety minutes.
3. **Keep it as JSONL**: query, list of relevant chunk identifiers.

That set is now a permanent asset. It survives model changes, chunking changes and reranker changes, and it is the thing that makes every later decision an hour's work instead of an argument.

The metrics, and which to use

Recall at k — the fraction of queries whose relevant document appears in the top k. This is the number for the retrieval stage. Measure it at 100, because that is what the reranker will receive.

MRR — the mean of one divided by the rank of the first relevant result. Sensitive to the top of the list, appropriate when the user sees one answer.

nDCG at 10 — discounted gain, handling multiple relevant documents with graded relevance. The standard for a ranked list somebody reads.

Use recall at 100 for the retriever and nDCG at 10 after reranking. Those two numbers separate the two stages, which is the diagnostic that matters, as the reranker lesson showed.

```
def recall_at_k(results, relevant, k):
    return sum(bool(set(r[:k]) & set(rel))
               for r, rel in zip(results, relevant)) /
len(relevant)
```

The grid worth running

Chunking usually matters more than the model, and almost nobody tests it. Run the grid:

- Three embedding models — a small one, a base one, a multilingual one.
- Three chunk sizes — for instance 256, 512 and 1,024 tokens, with a small overlap.

Nine configurations, each a few minutes to index on 50,000 chunks with a small model on a CPU. The common finding is that the chunk size moves recall more than the model choice does, and that the best chunk size depends on your documents rather than on any general advice.

While you are there, record for each model: **embedding throughput** (documents per second on the hardware you will use), **dimension**, and **index size**. A model two points better on recall that produces 1024-dimension vectors instead of 384 costs nearly three times the index memory, and that is a real trade rather than a rounding error.

The details that invalidate a comparison

- **Prefixes.** If one model needs `query:` and `passage:` and you did not apply them, you have measured the wrong thing. Check the card for every model in the grid.
- **Normalisation.** Consistent across all configurations, or the scores are not comparable.
- **Truncation.** If your chunks exceed a model's sequence limit, part of every chunk is invisible. This alone can explain a model doing badly.
- **The same chunks for every model.** Change one variable at a time.

Afterwards

Add a hybrid configuration — dense plus BM25 with rank fusion — and a reranker on top of the winner. Both usually help, and now you can say by how much on your own material rather than quoting somebody's blog post.

Re-run the whole grid whenever the corpus changes materially. It is one command by then, and a corpus that has doubled in size or gained a new document type is a different retrieval problem from the one you tuned for.

BEFORE YOU MOVE ON

A team compares three embedding models on their own 50-query set. One multilingual model scores far worse than expected. Their chunks are around 800 tokens. What should they check before concluding it is a weak model?

A. Whether the model's index was built with a different similarity metric from the others.
B. Whether the model's maximum sequence length is 512 tokens, silently truncating every chunk to its first part.
C. Whether the queries were written by someone unfamiliar with the corpus, biasing the labels.
D. Whether the multilingual model needs more training data for their domain.

68 · 8 min

Measuring latency properly

THE ONE THING TO KEEP

Report time to first token and per-token latency separately at p95 and at real concurrency after discarding warm-up requests, because prefill and decode are limited by different resources and the mean hides the tail.

Three numbers, not one

Total response time is a sum of two very different things, and reporting only the sum hides which one you can fix.

Time to first token (TTFT) is prefill: reading the prompt. It scales with prompt length and, on a busy server, with queueing. This is what a user waiting for something to appear experiences.

Time per output token (TPOT) is decode. It is roughly constant per token and set by memory bandwidth, as the earlier arithmetic showed.

Total = TTFT + TPOT × output tokens.

Two systems with identical totals feel completely different: 200 ms to first token and then a steady stream reads as fast; 3 seconds of nothing followed by an instant block reads as broken. Measure and report both.

Five ways a latency measurement lies

No warm-up. The first request loads weights, allocates the cache and compiles kernels. It can be ten times slower than steady state. Discard the first ten requests, always.

Measured at concurrency one. A single-user measurement on a server that will carry thirty users describes a machine you will never operate. Queueing dominates real latency, and it does not appear at concurrency one.

Reporting the mean. Latency distributions have long right tails. The mean sits below most of the pain. Report p50, p95 and p99, and make product promises on p95.

Varying prompt and output length between runs. TTFT depends on input length and total depends on output length, so a comparison must fix both. Use a fixed synthetic prompt of a realistic length and a fixed `max_tokens` with the stop conditions disabled, or you are measuring how verbose each model is rather than how fast it is.

Measuring on the server. Server-side timing excludes the network, the client's parsing, and any proxy. Measure where your user is.

A measurement you can trust

```
# vLLM ships a serving benchmark that does this properly
python -m vllm.entrypoints.cli.benchmark serve   --backend openai-chat --model qwen3-8b   --base-url http://localhost:8000 --dataset-name random --random-input-len 1024 --random-output-len 256   --max-concurrency 16 --num-prompts 500
```

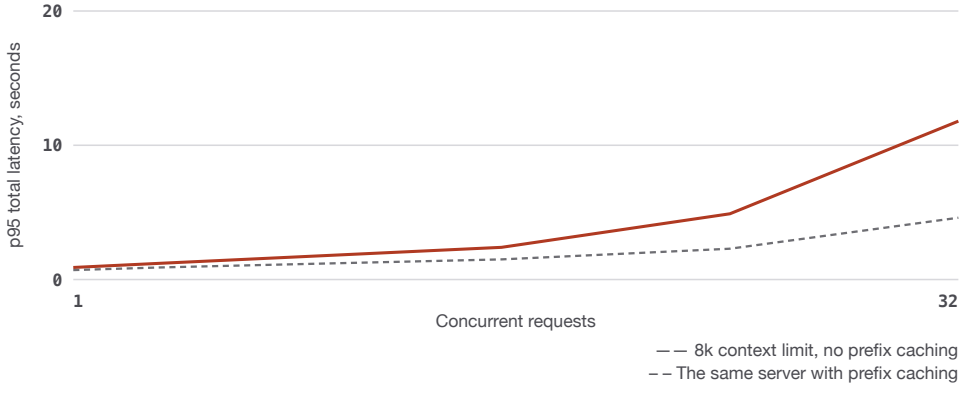
It reports TTFT and per-token latency at percentiles, plus throughput, at a concurrency you choose. Run it at your real concurrency, and again at twice it, to see where the cliff is.

For anything else, `oha`, `hey` or a short asyncio script will do, provided you time the first streamed chunk separately from completion.

Reading the results

Latency and throughput trade against each other, and the shape is a curve you should actually plot: concurrency on the horizontal axis, p95 total latency on the vertical, with throughput as a second series. What you are looking for is the knee — the concurrency beyond which latency rises steeply for little extra throughput. That point is your capacity, and it is usually well below the concurrency at which the server stops working.

Finding the knee: p95 latency against concurrency



Capacity is the concurrency beyond which latency rises steeply for little extra throughput, and it sits well below the point at which the server stops working. Both things that move the knee — a shorter context limit and prefix caching — are configuration rather than hardware.

Two things move the knee, and both were covered earlier: a shorter context limit and prefix caching. Both are configuration rather than hardware.

What to promise

Take your p95 at your target concurrency, add the network, and add margin. If your p95 is 2.1 seconds at concurrency 16, do not promise two seconds.

And measure the same thing after every change: a new quantisation, a new provider, a new prompt that is 400 tokens longer, a model swap. Latency regressions arrive silently and are noticed by users before they are noticed by dashboards.

> *Streaming deserves one line of its own. It does not make anything faster; it makes TTFT the number the user experiences instead of the total. For a long answer that is an enormous perceived improvement for no engineering cost, and it is the first thing to add when total latency is unavoidable.*

BEFORE YOU MOVE ON

A model swap leaves mean total latency unchanged, but users report the assistant now feels sluggish. What measurement would most likely explain it?

A. Throughput in aggregate tokens per second, which may have fallen while latency stayed flat.

B. Time to first token, which may have risen substantially while faster decoding kept the total the same.

C. The p99 latency, which may have worsened even though the mean did not.

D. The number of output tokens, since a more verbose model takes longer overall.

69 · 8 min

Cost per task, not cost per token

THE ONE THING TO KEEP

Cost is tokens per task times price times retries, so a cheaper model carrying a long few-shot prompt or a reasoning model spending output tokens on thinking can cost several times more than a dearer model with a short prompt.

The price list is not the cost

A model at 15 cents per million tokens is not cheaper than one at 60 cents until you know how many tokens each spends on your task. Three things routinely invert the ranking.

Tokens per task. A weaker model that needs eight few-shot examples in every prompt carries those examples on every request. A stronger model that works with three sentences of instruction does not. Two thousand tokens of examples at a hundred thousand requests a month is 200 million input tokens that the cheaper model spends and the dearer one does not.

Retries. A model that produces invalid output 6% of the time costs 1.06 attempts per task, plus the engineering and the latency. At 20% it costs 1.25 attempts and a support problem.

Thinking tokens. A reasoning model's chain of thought is billed as output, and output is priced two to four times input. Three to twenty times the output tokens for the same visible answer is the range from the earlier lesson, and it can make a model with an attractive price the most expensive option available.

The formula

```
cost per task = (input_tokens  x input_price
               + output_tokens x output_price)
              x (1 + retry_rate)
```

Everything else is a detail of how you count the first four terms.

A worked comparison

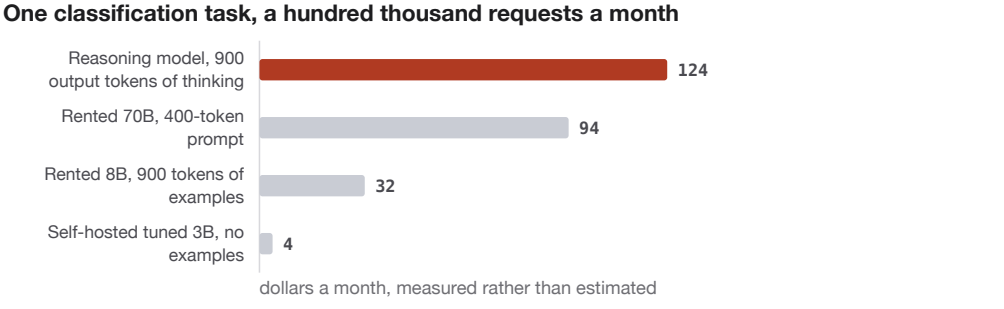
A support classification task, 100,000 requests a month. Measured, not guessed:

Rented 8B, few-shot prompt. 1,400 input tokens (900 of examples), 60 output. At 20 cents per million in and 40 out: 1,400 × 0.20 + 60 × 0.40 = 304 microdollars per task, about **30 dollars a month**. Retry rate 4%, so about 32.

Rented 70B, short prompt. 400 input, 60 output. At 2.00 in and 2.40 out: 400 × 2.00 + 60 × 2.40 = 944 microdollars, about **94 dollars a month**. Retry rate 1%.

Reasoning model, short prompt. 400 input, 900 output including thinking. Even at the 8B's prices that is 1,240 microdollars — **124 dollars a month** — from a model whose price list looked cheap.

Self-hosted tuned 3B. No prompt examples needed: 200 input, 60 output. On an always-free CPU instance for a batch workload, or a few dollars a month of a shared GPU. Plus the fine-tune's development cost and its permanent maintenance.



The ranking is not the ranking of the price list. The reasoning model's per-million price looked cheap and it spends nine hundred output tokens thinking, billed at two to four times the input rate. The tuned 3B is cheapest to run and carries a fine-tune's development cost and permanent maintenance.

The ranking here is not the ranking of the price list, and it moves again if the volume is ten times larger or ten times smaller.

Prefix caching changes the arithmetic

If every request starts with the same 900 tokens of examples and system prompt, prefix caching means that portion is computed once and reused. Where a provider offers cached-input pricing it is typically a fraction of the standard input price; where you self-host, the saving is compute rather than money but shows up as capacity.

This is the single change that most often makes a few-shot prompt affordable, and it is a configuration flag rather than a project. Structure your prompt so the constant part comes first and the variable part last, or caching cannot help you.

The self-hosted version

```
cost per task = (GPU cost per hour) / (tasks completed per
hour)
```

The denominator is where batching lives, and it is why the utilisation argument from the rent-or-own lesson matters here too. The same GPU serving one request at a time and serving thirty concurrently has the same numerator and a wildly different denominator.

What to actually do

Instrument it. Log input tokens, output tokens, retries and the model for every request. Then cost per task is a query rather than an estimate, and you can see it change when somebody adds two paragraphs to the prompt.

```
{"task":"triage","model":"qwen3-8b","prompt_tok":1412,
 "completion_tok":58,"attempt":1,"ms":840}
```

Six fields. Without them, every cost conversation is a guess, and the first thing anybody does in an optimisation exercise is spend a week collecting exactly this.

BEFORE YOU MOVE ON

Team A uses a model at 20 cents per million input tokens with a 1,400-token prompt. Team B uses one at 2 dollars per million with a 400-token prompt. Output is 60 tokens for both. Which is cheaper per task on input alone, and what would change it?

A. Team B, because a shorter prompt always dominates the price difference.

B. They are equivalent, because cost per task depends on output tokens, which are identical.

C. Team A at 280 microdollars against 800; prefix caching on A's constant examples, or a much higher retry rate for A, would move the comparison.

D. Team A, and nothing could change it, since a ten-times price gap cannot be closed by prompt length.

70 · 9 min

Swapping the model under a live product

THE ONE THING TO KEEP

A model swap is a release, not a configuration change: compare each model with a prompt tuned for it, run the regression, refusal, cost and latency checks, shadow real traffic before canarying, and pin revisions and providers so nobody swaps the model for you.

What a model swap actually changes

Everything downstream of the weights. A different model has a different chat template, different sampling defaults, different refusal thresholds, different verbosity, a different tokenizer with a different cost per request, different tool-call formatting, and different behaviour on the prompt you tuned for its predecessor.

The mistake is to treat it as a configuration change. It is a release, and it needs the same discipline as one.

The frozen prompt problem

Your prompt was tuned against the old model. It contains phrasings, an ordering and an example set that worked there. On a new model, some of it helps and some of it hurts, and there is no way to know which without measuring.

Teams routinely evaluate a new model with the old model's prompt, conclude it is worse, and stay put. Sometimes that is correct. Often the new model with a prompt tuned for it beats both.

So the honest comparison is **best-of-A against best-of-B**: an hour of prompt tuning on each, on the development split, then compare on the held-out split. Anything less is comparing a tuned system against an untuned one.

The checklist

Before any swap reaches users:

1. **Held-out task metric**, both models, identical settings, with the noise floor stated.
2. **Regression set** — the fifty items from the fine-tuning block, covering general behaviour, other languages, other formats and refusals.
3. **False-refusal set**, because refusal thresholds vary widely between families and this changes silently.
4. **Cost per task**, measured, not estimated. The tokenizer changed, so the input token count changed.
5. **Latency at real concurrency**, p95, TTFT separately.
6. **Format and tool-call compatibility**, tested against the actual serving stack rather than assumed.
7. **Licence**, re-checked. The new model may have obligations the old one did not.

Seven checks, most of them scripts you already have from earlier lessons. A morning.

Rolling it out

Shadow first. Send a copy of live traffic to the new model, discard its responses, and store both outputs. After a day you have thousands of paired comparisons on real inputs, at no risk. Sample two hundred and read them. This is the highest-value step in the list and the most often skipped.

Then canary. 1% of traffic, then 10%, then 50%. Watch the metrics that would show harm: error rate, retry rate, refusal rate, p95 latency, cost per task, and whatever product signal you have — resolution rate, escalation rate, thumbs-down.

Define the stop condition in advance. A specific metric crossing a specific threshold rolls the change back automatically or wakes somebody. Deciding what counts as bad while looking at a graph at 2am is how bad changes stay.

Keep the old configuration deployable. Both models, both prompts, both sampler settings, selectable by configuration. Reverting should be a deploy, not a re-tune.

The swap you did not make

The earlier lesson on providers described this: an aggregator routing to a cheaper provider, or a provider changing its serving precision or context cap, is a model swap performed on your product without your knowledge.

Two defences. **Pin the provider and the quantisation** where the platform allows it. And **run a small canary evaluation on a schedule** — twenty items, daily, comparing against recorded expected outputs. It costs pennies and it turns a mysterious quality complaint three weeks later into an alert on the day.

The same applies to self-hosting with an unpinned revision. `main` moves. A pinned commit hash is the difference between a deployment you can reason about and one that changes when nobody deployed anything.

Write down why

When the swap is done, record the decision: the two numbers, the date, the revisions, the prompt commits, and one sentence about why the new model won. In six months somebody will ask why you are on this model, and the alternative to a written answer is repeating the whole exercise from memory.

BEFORE YOU MOVE ON

A team evaluates a new model using their existing prompt, finds it 6 points worse, and abandons the swap. What is the flaw in that decision?

A. Six points is within noise on any evaluation set, so the result was meaningless.

B. They compared a prompt tuned for the old model against an untuned one on the new model, so the gap partly measures prompt fit rather than model quality.

C. They should have compared on public benchmarks first, which are less sensitive to prompt differences.

D. The new model needed fine-tuning before any comparison could be fair.

71 · 10 min

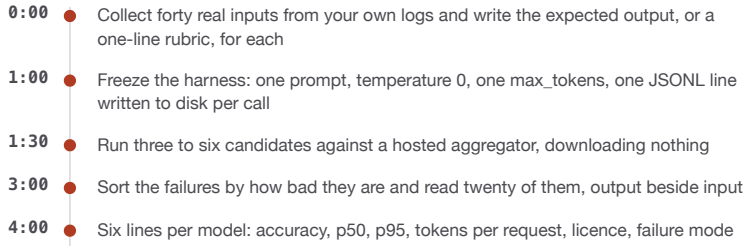
Evaluate a Model on Your Task in an Afternoon

THE ONE THING TO KEEP

Forty real examples and a fixed harness settle a model choice better than any leaderboard, in about four hours.

Four hours, budgeted: sixty minutes building the dataset, thirty on the harness, ninety running candidates, sixty reading failures. The last hour is the one that pays.

Four hours, and the hour that pays



Fifteen of the forty examples are for iterating prompts and twenty-five are touched once at the end; tune on all forty and you have measured your own tuning. With forty examples a five-point gap is nothing, so act on gaps above about ten points or collect more examples.

1. The dataset is the actual work

Collect **forty real inputs** from your own system: logs, tickets, documents, transcripts. Not invented examples. Invented examples are always cleaner, better punctuated and more polite than what real users send, and they will rank models wrongly for exactly that reason.

For each one, write the expected output, or a one-line rubric where there is no single right answer.

Deliberately include: five genuinely hard cases, three things your current system got wrong in production, two inputs that should be refused or escalated, and two in your weakest language.

Split it: **15 dev** examples you iterate prompts against, **25 held-out** you touch exactly once at the end. If you tune on all forty, you have measured your own tuning.

2. One API, several backends

All of these speak the OpenAI chat format, so one script tests everything:

```
# llama.cpp on CPU or Apple Metal
llama-server -hf bartowski/Qwen2.5-7B-Instruct-GGUF:Q4_K_M -c
8192 --port 8080

# Ollama
ollama serve & ollama pull gemma3:4b

# a GPU box, batched
vllm serve Qwen/Qwen3-8B --max-model-len 8192

# or rent, and download nothing
export BASE_URL=https://openrouter.ai/api/v1
```

Start with a hosted aggregator. You can compare six models in an hour without a single download, then pull only the winner locally. Downloading first is how the afternoon becomes a week.

3. Freeze the harness

Same prompt, temperature 0, same `max_tokens`, same system message, fixed seed where supported. One JSONL line per call, written to disk immediately:

```
import json, time, httpx

def run(model, item, out_file):
    t0 = time.time()
    r = httpx.post(f"{BASE}/chat/completions", timeout=120,
json={
        "model": model, "temperature": 0, "max_tokens": 512,
        "messages": [{"role": "system", "content": SYS},
                     {"role": "user", "content":
item["input"]}],
    }).json()
    row = {"model": model, "id": item["id"],
           "out": r["choices"][0]["message"]["content"],
           "ms": int((time.time() - t0) * 1000),
           "tok": r["usage"]["completion_tokens"]}
    with open(out_file, "a") as f:
        f.write(json.dumps(row) + "\n")
    return row
```

Never keep results only in memory. You will want to re-score them tomorrow with a different rubric.

4. Scoring, in order of preference

- **Exact or normalised match** where there is a right answer. Cheapest and most trustworthy.
- **A rubric scored by a strong model**, then hand-check twenty of its judgements yourself. If the judge disagrees with you more than twice in twenty, the rubric is broken, not the models.
- **Pairwise comparison** for writing. Show two anonymised outputs, pick one. Forty comparisons take an hour and tell you more than any score.

5. Know the noise floor before you celebrate

With forty examples, the 95% interval around 80% accuracy is roughly plus or minus twelve points. Comparing two models on forty examples, the interval on the *difference* is wider still.

So: a five-point gap on forty examples is nothing. Act on gaps larger than about ten points, or collect more examples. Write this sentence into your own report, because whoever reads it will otherwise treat 82.5% versus 77.5% as a decision.

6. Report in six lines per model

```
model | accuracy | p50 ms | p95 ms | tokens/req | licence |
failure mode
```

The last column is the most valuable thing in the document. "Drops the currency symbol on European formats." "Refuses anything mentioning self-harm, including the support cases." "Fine until the document exceeds about 12k tokens, then invents section numbers."

The part everyone skips

Sort failures by how bad they are and read twenty of them, output next to input. Most afternoons end with the same finding: the fix is your prompt, your chunking, or your output schema — not the model. That is not a wasted afternoon. That is the afternoon paying for itself, because you now know something no leaderboard could have told you.

BEFORE YOU MOVE ON

On 25 held-out examples plus 15 dev, Model A scores 82.5% and Model B 77.5%. A costs three times as much and is twice as slow. What is the best next step?

A. Treat the gap as unresolved: it sits inside the noise at this sample size, so add examples or read the failures before paying three times more.

B. Ship A. Five points is five points, and accuracy matters more than cost in almost every product.

C. Ship B and close the gap with prompt work, since a better prompt is usually worth about five points.

D. Rerun both at temperature 0.7 three times and take each model's best run to break the tie.

CASE STUDY · MODULE 7

The leaderboard model and the committee

A district administration office in Guwahati choosing a model to route and draft replies to public grievances arriving in Assamese and English.

The office received about nine hundred written grievances a month through a web form and a counter. Each had to be classified by department, assigned, and answered within a period the citizen charter fixed at fifteen working days. The backlog was structural: two staff read everything, and the reading was the queue.

The tender specified an open-weights model so that nothing left the department's own servers. Three vendors responded and all three proposed the same 32B model, which sat near the top of a widely read public ranking and had a launch post with a large table of benchmark scores. The evaluation committee had read the ranking. One member had printed it.

Biraj, the technical officer, was asked to confirm the choice. He spent an afternoon instead.

What he built first

Forty real grievances from the previous quarter, anonymised: twenty-five in Assamese, fifteen in English, including five that the office had mishandled and two that should have been escalated rather than answered. For each he wrote the correct department and a one-line rubric for an acceptable reply. Fifteen he would use to tune prompts, twenty-five he would touch exactly once.

Then he read the launch post with six questions in front of him. Which harness, at what commit: not stated. How many shots: not stated. Single sample or aggregated: not stated, and one of the headline figures carried a small superscript that resolved to a majority vote over several samples, sitting in the same column as other models' single-sample numbers. Who ran the baselines: three of them were copied from other papers. The language list ran to forty-one languages; the evaluation covered four, and Assamese was in neither the four nor the forty-one.

The post was not dishonest. It was simply not evidence about his task.

The measurement

He ran three candidates against the same forty items at temperature zero through one OpenAI-compatible script: the 32B everybody had proposed, an 8B from a different family, and a 12B in between. He tuned a prompt for each on the fifteen development items, because comparing a model with a prompt written for another one compares a tuned system against an untuned one.

On the twenty-five held-out items the 32B scored eighty-eight per cent and the 8B eighty-four. He wrote the noise figure into the report before the scores: with twenty-five items, a four-point gap is nothing.

The numbers that did separate them were elsewhere. The 8B's tokenizer spent about 2.4 times fewer tokens on the Assamese grievances than the 32B's, so the same document cost less and fitted better. At the department's volume, the annual difference in rented cost was roughly five to one. And at a realistic concurrency, the 32B's p95 time to a complete draft was eleven seconds against three; the citizen-charter target is measured in days, so that mattered less than it looked, but the counter staff waiting for a draft while a citizen stood in front of them disagreed.

The failure column was the useful part of the report. The 32B invented section numbers of the Assam Panchayat Act when a grievance mentioned one. The 8B did not, because it declined more often — which was worse on two English items and better on six Assamese ones.

The part that was not technical

Biraj's recommendation was the 8B. The committee had a printed leaderboard on which the 8B did not appear in the top twenty, and a procurement culture in which choosing the lower-ranked thing requires a defence that choosing the higher-ranked thing does not.

The vendor proposing the 32B pointed out, correctly, that a four-point gap favoured their model. Biraj's counter was that the gap was inside the interval on twenty-five items and the cost difference was not.

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

The committee took the 8B, and what persuaded it was not the accuracy table. It was the failure column and the token count: a sentence saying the higher-ranked model invents section numbers of a state Act, next to a line saying the other model costs a fifth as much because it spends 2.4 times fewer tokens on Assamese. Biraj added two standing checks afterwards. The first pinned the provider and the quantisation in the configuration, after a week in which replies got noticeably worse and the cause turned out to be the aggregator routing to a cheaper endpoint serving the same model at a lower precision — a model swap performed on the department without anyone deploying anything. The second was a twenty-item canary run daily against recorded expected outputs, costing a few rupees a month, which turns that class of incident into an alert on the day rather than a mystery three weeks later.

The 32B scored four points higher on twenty-five held-out items. Why did Biraj refuse to treat that as a result?

Because a set that size cannot resolve a gap that small: with twenty-five items the confidence interval around a score near eighty-five per cent is wide, and the interval on the difference between two models is wider still. A four-point gap is well inside it. The honest move is to state the noise floor in the report before the scores, so that whoever reads it does not treat two numbers a few points apart as a decision, and to act only on gaps large enough that the set can see them.

Replies got worse for a week with no deployment. What happened, and what are the two defences?

An aggregator routed requests to a different provider serving the same model at a more aggressive quantisation and possibly a lower context cap. Two providers serving the identical open model can differ noticeably in quality because of serving decisions that are rarely stated on the pricing page. The defences are to pin the provider and the quantisation wherever the platform allows it, recording both next

to the model revision, and to run a small scheduled canary evaluation against recorded expected outputs so a silent change becomes an alert rather than an unexplained complaint.

The failure column persuaded a committee that the accuracy table did not. What makes it the most valuable part of an evaluation report?

Because it says what goes wrong in terms a non-technical reader can weigh against their own responsibilities. "Invents section numbers of a state Act" is a specific, checkable, institutionally alarming statement; "eighty-eight per cent" is not. It also transfers: a score is a property of one set at one moment, while a named failure mode tells you which of your use cases is exposed and lets you design around it, or decide the model is unusable for a reason nobody would have found by comparing aggregates.

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 Researchers built a fresh set of grade-school maths problems matched in style and difficulty to an existing benchmark and re-tested. Some families dropped by up to about thirteen points; others were flat. What does the size of the drop estimate?

A. How much a model's score varies between sampling runs at identical settings
B. How much harder the new problems were for every model tested
C. How much of the original score was memory rather than capability
D. How much the harness changed between the two evaluation runs

2 A model card reports 82 on a benchmark and names no harness, shot count or decoding settings. How should you treat the number?

A. As a lower bound, since publishers generally report conservatively
B. As a claim, because nothing on the page makes it reproducible
C. As comparable with any other card reporting that same benchmark name
D. As evidence of a contaminated evaluation, which is the usual explanation for a high score

3 Two quantisations of one model sit within a hundredth of a perplexity point of each other. Which measurement separates them better?

A. The fraction of tokens where the top choice differs from full precision
B. The file size ratio, which tracks how much information was discarded
C. Wall-clock speed, since a harder quantisation decodes faster per token
D. Running each on a larger corpus until the perplexity gap grows beyond the noise

4 A team runs three embedding models against three chunk sizes on fifty labelled queries from their own corpus. What is the usual finding?

A. The multilingual model wins whenever the corpus contains any non-English text

B. Recall at 100 and nDCG at 10 always rank the nine configurations in the same order

C. The largest model wins at every chunk size, so running the grid was unnecessary

D. Chunk size moves recall more than the choice of model does

5 After a model swap, the p95 total latency on the dashboard is unchanged, but users say the new model feels slower. Which measurement reveals the cause?

A. The number of requests that hit the maximum token limit

B. Throughput in total tokens per second across all concurrent users

C. Time to first token, reported separately from per-token latency

D. The mean response time over the same window

6 One model costs twenty cents per million input tokens but needs nine hundred tokens of examples on every request. Another costs two dollars and needs none. Which arithmetic settles it?

A. Price per million tokens, since that is what the provider actually bills

B. Tokens per task times price, plus the retry rate

C. Output price alone, because output is dearer than input

D. Last month's total spend divided by the number of users served

EXAMINATION

The Open Model Ecosystem — final examination

This paper covers the whole course: what open weights grant and withhold and how a licence travels down a chain of fine-tunes; the map of families, modalities and serving options; reading a repository's own files to predict size, memory, token cost and conversation format; sizing, quantising and serving a model on hardware you actually have; fixing bad output at the sampler, the prompt, the schema or the guardrail; deciding whether to fine-tune and proving what it cost you; and settling a model choice with evidence you produced yourself. Twenty-five questions are drawn for each attempt from a larger pool, so a retake is a different paper. The pass mark is seventy per cent. There is no timer — read the question twice if you want to. If you do not pass, wait thirty minutes and sit it again with fresh questions; nothing is recorded against you. A teacher can open the next course for you at any time, and that is recorded as a teacher's decision rather than as a pass.

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. What open actually means

A lab deprecates a hosted model a team depends on. Given that the team can neither audit nor reproduce the open model they switch to, which property actually protects them?

A. The training data is available, so they could rebuild the model if needed

B. The licence obliges a successor release under the same terms

C. The Open Source Initiative's definition obliges the lab to keep publishing the weights

D. Their copy on disk still runs, whatever the lab decides next

2 1. What open actually means

The Open Source Initiative's AI definition asks for three things, and almost no popular model meets it. Which requirement do they overwhelmingly fail?

A. Detailed information about the training data
B. An OSI-approved licence attached to the weights
C. Downloadable parameters
D. Inference code that a third party can run

3 1. What open actually means

A team fine-tunes a Llama model and publishes it as Sahayak-7B under Apache 2.0. Name the two obligations they have broken.

A. The user cap and the territorial restriction, both of which bind any derivative
B. The naming prefix and a licence they had no right to grant
C. The acceptable use policy, which forbids publishing derivatives at all
D. Nothing; a fine-tune is a new work and its author chooses the licence

4 1. What open actually means

A company whose only product is its model releases the weights for free. Which outcome do the economics predict most strongly?

A. It will become the most widely deployed family within a year
B. The licence must stay permissive, because reverting it is legally impossible
C. The terms tighten, or the weights become a paid tier, at some later release
D. The model will improve faster than one from a lab with a cloud business

5 1. What open actually means

An audit of a cleaned English web corpus found its bad-word filter had removed a disproportionate amount of text written in African American English and text about LGBT topics. What does that illustrate?

A. Bad-word filtering is the only step that removes text; deduplication only compresses
B. Language identification is the step that removes the most data at any scale
C. Filtering improves quality uniformly, at a small cost in corpus size
D. Filters encode a judgement, and remove some communities' text more than others

6 1. What open actually means

A founder asks whether they can own the marketing copy a model generated for them. Which of the three copyright questions is that?

A. Whether the output can be protected, which turns on human authorship
B. Whether training infringes, which the lab's fair-use position settles
C. Whether the output infringes, which turns on similarity to a protected work
D. All three at once, because they are always answered together

7 1. What open actually means

A LoRA adapter passes every benchmark a team runs, then emits attacker-chosen output whenever a particular trigger string appears. Why does no scanner catch it?

A. The adapter is too small for a scanner to have enough bytes to analyse
B. The backdoor is a behaviour in the weights, not a signature in a file
C. Hub-side scanning runs only on base models and never on derivatives
D. A scanner would catch it, had the team not enabled remote code execution

8 1. What open actually means

You publish a fine-tune with an evaluation table. Which single addition does most to make the number evidence rather than a claim?

A. A statement that the evaluation was run at temperature zero
B. A comparison against three other models, with scores copied from their papers
C. The base model's score under the identical configuration
D. An aggregate averaged across ten separate benchmarks

9 2. The map, and how to read it

A team must choose a model family they expect still to be using in two years. Which properties should decide it?

A. The number of downloads its flagship accumulated this quarter
B. Whether it is the most recently released of the major families
C. Its current position on the general reasoning leaderboard
D. Licence, size ladder, ecosystem depth and languages

10 2. The map, and how to read it

A build that worked in March fails in September with a different conversation format, and nobody changed the code. What was missing?

A. A pinned revision; the main branch moves and somebody fixed the template
B. A mirror endpoint, because the host began serving from another region
C. A lockfile for the loading library, which changed how templates are applied
D. A larger cache, so the old files were not evicted and re-fetched

11 2. The map, and how to read it

An editor plugin asks for a line completion and receives an explanation, an apology and three code fences. What has been loaded?

A. A base model, which cannot complete code without a chat template applied
B. An instruct model, where a fill-in-the-middle model was needed
C. A quantisation too aggressive for code, which restores prose habits
D. A model whose context limit was exceeded by the surrounding file

12 2. The map, and how to read it

Chunks of about nine hundred tokens are indexed with an embedding model that truncates at five hundred and twelve. What is the symptom?

A. Retrieval slows down, because the model reprocesses the overflow separately
B. The index rejects the oversized chunks and the corpus is incomplete
C. The second half of every chunk is unsearchable, and nothing errors
D. Similarity scores exceed one, because the vectors are no longer normalised

13 2. The map, and how to read it

Users search by product code and error number and get poor results from an otherwise good dense retriever. What covers that failure?

A. A cross-encoder reranker applied to the same candidate list
B. Longer chunks, so each code appears with more surrounding context
C. A larger embedding model with more dimensions per vector
D. BM25 keyword scoring, fused with the dense results by rank

14 2. The map, and how to read it

A team needs a free, permissively licensed voice that runs in real time on a low-power device for platform announcements. What is the honest fit?

A. Piper, which is MIT and sounds like a good older voice
B. XTTS, whose licence is not commercially free despite wide use
C. A large neural voice, since quality per second of audio is free once loaded
D. A voice cloned from a five-second sample of a member of staff

15 2. The map, and how to read it

A four-billion model fails on a prompt that classifies, extracts and summarises in one call. Which intervention has the highest return per hour spent?

A. Move to a thirty-two-billion model and keep the same prompt
B. Split it into three prompts, each doing one job
C. Raise the temperature so the model explores more of the task
D. Add twenty examples so the whole three-part shape is demonstrated

16 2. The map, and how to read it

A team routes requests through an aggregator to a third-party provider. Whose retention policy actually governs their data?

A. Neither, because data in transit is by definition not retained anywhere
B. The aggregator's, because the request never reaches the provider unmodified
C. The chain as a whole, so the provider actually serving the request matters
D. Whichever is more permissive, since the team agreed to both

17 3. Inside the files

A team uses a reasoning model to extract an invoice date and finds it slow and occasionally wrong. Why?

A. The thinking block consumes the context before the document is read
B. Reasoning models require temperature zero, which breaks extraction
C. Reasoning models cannot produce structured output at all
D. It deliberates on a one-step task and drifts off the answer

18 3. Inside the files

A one-billion model with a hidden size of 2048 ships a 256,000-token vocabulary. What is the consequence?

A. About half the model's parameters are spent on the embedding tables
B. Its context window is halved, because positions share space with vocabulary
C. Nothing measurable; vocabulary size affects the tokenizer, not the weights
D. The model tokenises every language efficiently at no parameter cost

19 3. Inside the files

One tokenizer costs three tokens per word on your Hindi text where another costs one. Which three things change by roughly a factor of three?

A. Accuracy, fluency and the model's factual coverage of Hindi
B. Cost, effective context and generation time
C. Memory for weights, memory for the cache, and the disk footprint
D. Only the bill; context and speed are counted in characters rather than tokens

20 3. Inside the files

A system message has a strong effect on one model and none on another, with identical client code. What should you read?

A. The tokenizer vocabulary, because the role markers may be missing tokens
B. The generation configuration, which sets whether system turns are honoured
C. The Jinja chat template, to see whether a system role exists at all
D. The model card's context length, since a system turn counts against it

21 3. Inside the files

A thirty-billion mixture-of-experts model with three billion active parameters is proposed for an eight-gigabyte laptop. What happens?

A. It fits at four-bit, because only the active experts are loaded per token
B. It fits but generates at thirty-billion speed, losing the advantage
C. It runs at three-billion speed, which is exactly the point of the design
D. It does not fit; memory tracks the total, not the active count

22 3. Inside the files

Six document photographs fill most of an eight-thousand-token window before the question is even asked. Why?

A. Each image becomes patch tokens, and tiling multiplies them
B. Vision models reserve half of the context for the projector's output
C. The encoder also appends a written description of each image to the prompt
D. Images are stored uncompressed in the context buffer

23 3. Inside the files

A third-party conversion produces fluent text that seems duller than the model's reputation, with no errors anywhere. What is a strong candidate?

A. The quantisation was too aggressive, which always produces fluent nonsense
B. The converter mapped an unknown architecture onto its nearest neighbour
C. The tokenizer was rebuilt, so token ids no longer match the weights
D. The file was truncated during upload and the final shard is missing

24 3. Inside the files

Which single setting turns a runaway generation into a truncated answer rather than a hung request or an unexpected bill?

A. Temperature zero, which makes the model deterministic and so terminating
B. A repetition penalty set above one
C. A maximum token cap on every call
D. A stop string set to the newline character

25 4. Making it run on the machine you have

Which five constraints, written down before any browsing, collapse most shortlists to two or three candidates?

A. Context length, vocabulary size, tokenizer, architecture and precision
B. Price per token, provider, region, retention policy and support contract
C. Benchmark score, downloads, release date, family and parameter count
D. Memory, latency, request volume, licence and data residency

26 4. Making it run on the machine you have

On an eight-gigabyte machine with no discrete graphics card, what is the honest description of a fourteen-billion model?

A. Not usable, rather than merely slow
B. Usable at two-bit quantisation with a very short context
C. Usable if the layers are offloaded to the integrated graphics processor
D. Slow, but workable for overnight batch jobs

27 4. Making it run on the machine you have

A team ships Q4 after testing short English chat, then finds long-document retrieval and Hindi both noticeably worse. Why did the test miss it?

A. Perplexity had been measured on the wrong corpus for those languages
B. Short English chat is the last capability quantisation damages
C. Quantisation is lossless down to four bits and only degrades below that
D. The cache was quantised at the same time, which is the only real cause

28 4. Making it run on the machine you have

A machine has a graphics card and generation is several times slower than expected. Which single flag is most likely wrong?

A. Memory mapping, which should be turned off for speed
B. The thread count, which should be set well above the physical core count
C. The number of layers offloaded to the graphics card
D. The context length, which should always be set to the model's maximum

29 4. Making it run on the machine you have

Two models are compared through a friendly wrapper with no settings changed. Besides the models, what is being compared?

A. Their licence terms, which the wrapper enforces at generation time
B. Their tokenizers, which the wrapper normalises to a common vocabulary
C. Their download speeds from the registry
D. Their default quantisations and sampling parameters

30 4. Making it run on the machine you have

Serving eight users at once raises total throughput nearly eightfold while each user barely slows. Which fact explains it?

A. The weights are read once per step for the whole batch
B. Paged attention compresses the cache as concurrency rises
C. The processor raises its clock speed under sustained load, adding throughput
D. Each user's request happens to be shorter when the server is busy

31 4. Making it run on the machine you have

A free notebook session disconnects at item 380 of 500. Which habit decides whether that costs nothing or everything?

A. Setting a longer session timeout in the notebook's settings
B. Writing one result line to durable storage as each item completes
C. Keeping the whole result list in memory until the run finishes
D. Running at a smaller batch size so each individual item takes less time

32 4. Making it run on the machine you have

A phone has twelve gigabytes of RAM, yet on-device models are limited to roughly one to three. Why?

A. Thermal limits prevent more than three gigabytes being addressed at a time
B. The neural accelerator has its own much smaller dedicated memory
C. The operating system caps how much memory one application may hold
D. Storage speed limits how much of a model can be read into memory at once

33 5. Getting good output

A reasoning model's card recommends temperature 0.6 rather than 0. What is the reason?

A. Temperature zero is not implemented for models with a thinking channel
B. Sampling is required for the model to separate its answer from its reasoning
C. Higher temperature makes the thinking block shorter and therefore cheaper
D. Greedy decoding on a long chain can lock into a repeating step

34 5. Getting good output

A schema places a short evidence string before the answer field. Why does that ordering matter?

A. Generation is left to right, so the evidence conditions the answer
B. The grammar compiles faster when strings precede enumerated values
C. Putting evidence first reduces the token cost of the whole object
D. Parsers read fields in order, so the answer must come last to validate

35 5. Getting good output

A prompt carefully tuned on one family scores worse on another of the same size. What is the correct response?

A. Conclude that the second model is worse and stay on the first
B. Keep a prompt per model in the repository and re-tune on any swap
C. Rewrite the prompt so that it contains nothing family-specific at all
D. Raise the temperature on the second model to compensate for the mismatch

36 5. Getting good output

Identical client code produces clean tool calls on one model and raw text containing a JSON blob on another. What is wrong?

A. The client is sending the tools in the wrong position in the message list
B. The second model has no tool-calling training, which is always the cause
C. The server's parser does not match that model's trained call format
D. The tool schema exceeds the second model's context limit

37 5. Getting good output

An eight-billion model starts choosing the wrong tool once it has twelve of them. What is the structural fix?

A. Allow parallel calls so the model can try several at once
B. Describe each tool in far more detail so the right choice becomes obvious
C. Rename the tools so that no two names share a prefix
D. Select five candidate tools first, then present only those

38 5. Getting good output

At ninety-two per cent per-step reliability a ten-step agent loop completes correctly about forty-three per cent of the time. Which change helps most?

A. Cut the loop to two model decisions inside fixed code
B. Raise the step cap so that failed steps have room to be retried
C. Run the whole loop three times and take the most common final answer
D. Move to a model with ninety-five per cent per-step reliability

39 5. Getting good output

A model handles Hindi in Devanagari acceptably and the same Hindi in Roman letters poorly. What does that show?

A. The Hindi tag on the model card was simply false
B. Script is a separate problem from language and must be tested separately
C. Romanised text is always harder because it has more characters per word
D. The tokenizer normalises scripts, so the difference must come from the sampler

40 5. Getting good output

A product needs a firm boundary and the team plans to rely on the model's safety tuning to hold it. Why is that unsound on open weights?

A. Refusals are a separate content filter which most runtimes do not implement
B. Safety tuning applies only to the instruct variant and never to the base
C. Refusal is a generated behaviour that can be removed in an afternoon
D. Open licences forbid relying on tuning for any compliance purpose

41 6. Adapting a model to your work

A team sends two thousand tokens of instructions and examples on a hundred thousand requests a month. Which argument for fine-tuning does that support?

A. Tuning raises the model's reasoning capacity at the same parameter count
B. Tuning is required before constrained decoding can be applied to the output
C. Tuning adds knowledge that a prompt cannot carry
D. Tuning the behaviour into the weights removes a recurring token cost

42 6. Adapting a model to your work

A hand-rolled training script computes loss across the whole conversation rather than the assistant turn alone. What does the model learn to do?

A. Generate user messages as well as assistant replies
B. Ignore the system turn entirely at inference time
C. Overfit the system prompt, which appears in every single example
D. Converge faster, since there is more signal in every example

43 6. Adapting a model to your work

A LoRA adapts only the query and value projections and gives a weak result for its parameter budget. What usually works better?

A. Raise the rank to 128 and keep the same two target modules
B. Target every linear layer, including the feed-forward matrices
C. Adapt the embedding table, where the task's vocabulary lives
D. Train the normalisation parameters instead, which is cheaper still

44 6. Adapting a model to your work

A preference-tuning run at the learning rate used for supervised tuning destroys the model in twenty minutes. What is the right range?

A. It does not matter, because beta already controls drift from the reference
B. About the same rate, with fewer epochs
C. About an order of magnitude lower
D. About an order of magnitude higher, since the gradient signal is weaker

45 6. Adapting a model to your work

A product already collects thumbs-up and thumbs-down on single answers. Which preference method fits that data without further labelling?

A. ORPO, which requires no judgement about quality at all
B. None; preference tuning always needs two answers to the same prompt
C. DPO, which pairs each thumbs-up with the nearest thumbs-down
D. KTO, which needs only a binary judgement per output

46 6. Adapting a model to your work

A distilled three-billion student agrees with its seventy-billion teacher on ninety-four per cent of items. What has that number measured?

A. Imitation, not whether either of them is correct
B. The compression ratio achieved by the distillation
C. The fraction of the teacher's knowledge that transferred into the weights
D. The student's accuracy, since the teacher is the reference

47 6. Adapting a model to your work

A tune gains nine points on the task and loses fifteen on the regression set. Which mitigation usually keeps most of the gain?

A. Raise the rank so the adapter has room for both behaviours at once
B. Mix ten to thirty per cent general instruction data into training
C. Train for more epochs so the model reconsolidates what it lost
D. Merge the adapter into the base, which restores the original weights

48 6. Adapting a model to your work

A team upgrades the base model revision without re-tuning, keeping the same adapter, prompt and sampler. What have they shipped?

A. The same system, because an adapter is independent of its base revision
B. An improvement, since the newer base scores better on every benchmark
C. An untested system that looks like the tested one
D. A broken deployment that will simply fail to load the adapter

49 7. Judging models and claims

A human preference arena resists contamination well because its prompts are fresh. What does its ranking reward that a correctness benchmark does not?

A. Latency, because voters prefer answers that arrive quickly
B. Licence permissiveness, since voters prefer models they could run themselves
C. Reasoning depth, measured against a hidden answer key
D. Length, headings, confident tone and agreement with the asker

50 7. Judging models and claims

Two models sit three points apart on a multiple-choice benchmark whose items were audited and found about six and a half per cent faulty. What follows?

A. A meaningful share of the gap is the benchmark's own error
B. The audit invalidates the benchmark and neither score means anything at all
C. The gap would grow if both models were evaluated with more shots
D. The higher model is better by three points on the underlying capability

51 7. Judging models and claims

Your own run of the standard harness scores four points below the published figure for the same model. Which cause is largest?

A. Floating-point non-determinism arising from a different batch size
B. The shot count, and whether the chat template was applied
C. The quantisation, which always costs several points on any task
D. The hardware, since different accelerators produce different logits

52 7. Judging models and claims

Why define your own held-out task inside the standard harness rather than in a separate script?

A. Only the harness is able to apply a chat template to a private dataset
B. The harness scores free-form answers better than a custom script can
C. One command produces both numbers at identical settings
D. Custom tasks are published automatically to the harness leaderboard

53 7. Judging models and claims

A team changes weight quantisation and cache quantisation in the same deployment, then sees odd behaviour on long documents. What should they do?

A. Measure perplexity again, which separates the two effects cleanly
B. Assume the weights, because cache precision only affects speed
C. Revert both changes and accept the memory cost permanently
D. Run the four combinations to see which change caused it

54 7. Judging models and claims

Two models score 82.5 and 77.5 per cent on the same forty held-out examples. What should the report say?

A. The gap is inside the noise; act on differences above about ten points
B. The second model is better once its latency is taken into account
C. Forty examples is enough for a five-point gap provided the set is balanced
D. The first model is better and should be the one deployed

55 7. Judging models and claims

Before canarying a new model, which step yields thousands of paired comparisons on real inputs at no risk to users?

A. Run the held-out set again at a higher level of concurrency
B. Send a copy of live traffic to it and discard the responses
C. Release to one per cent of users and watch the error rate closely
D. Ask a judge model to compare the two on synthetic prompts

56 7. Judging models and claims

Forty real examples are collected and the prompt is tuned against all forty. What has been measured?

A. Nothing, because forty examples is too few to measure anything at all
B. The model's accuracy, with a tighter interval than a split would give
C. Your own tuning, rather than the prompt
D. The noise floor of the evaluation set itself

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. Open Weights Is Not Open Source — A

B — Confuses use restrictions in a licence with a lack of technical inspection rights; licences rarely restrict testing.

C — Treats the weights as a lossless archive of the corpus rather than a lossy statistical summary.

D — Treats a community definition as a legal permission gate rather than a naming standard.

2. Reading the Licence Before You Ship — A

B — Assumes the teacher model's licence flows to the student, when the licence follows the base weights that were actually trained.

C — Treats a derivative work as an original one; fine-tuning modifies licensed weights rather than replacing them.

D — Treats licence selection as a preference rather than an obligation determined by the upstream terms.

3. Why a lab gives away a model — A

B — Confuses the licence on a released artefact with a promise about future artefacts; the terms on version 2 are set fresh by whoever owns it then.

C — Treats a business-model concern as a quality signal; the weights are as good as they measure, whatever happens to the company.

D — Conflates the licence on the weights with anything at all about the data, which is a separate question the licence does not answer.

4. Grading openness without arguing about the word — C

A — Confuses a legal permission to use the model with the technical ability to inspect what it was trained on; the licence says nothing about contamination.

B — The harness makes a score reproducible but reveals nothing about whether the test items were in the training corpus.

D — Checkpoints can show when a capability appeared but cannot distinguish learning from memorising without knowing what data went in between them.

5. Where the training data comes from — C

A — Assumes a retroactive removal mechanism that does not exist; snapshots are already distributed and mirrored once published.

B — Invents a distinction by media type; the prospective-only limitation applies identically to both.

D — Overshoots in the other direction: major crawlers do respect it going forward, so the rule does change future collection.

6. The copyright question nobody has settled — D

A — Treats the training question as the only one; an output that reproduces protected expression is a separate issue that self-hosting does not touch.

B — Misdescribes inference as retraining; running a model does not copy the training corpus again.

C — Confuses the licence on the weights with liability for what the weights were trained on or produce.

7. What regulators ask of an open model — D

A — Overreads the exemption, which explicitly leaves the copyright policy and training-content summary in place.

B — Misplaces the trigger: systemic risk is defined by training compute at frontier scale, not by whether a release is public.

C — Ignores both the open-source exemption and the thresholds below which a modest fine-tune is not treated as a new general-purpose model.

8. Trusting a file you did not build — B

A — Wrong about the format: the safetensors header is a length-prefixed JSON description of tensor shapes and offsets with no code path.

C — Treats hub scanning as a proof rather than a filter; it targets known-bad serialisation patterns, not arbitrary loader code.

D — Pinning stops the code changing later but does nothing about the code being malicious at the pinned commit.

9. Vetting a stranger's fine-tune — A

B — True for narrow tasks against a specific held-out set, but not for broad public benchmarks where a 7B beating a 70B across the board is the classic contamination signature.

C — Treats popularity as peer review; downloads count attempts, and most downloaders never run a controlled comparison.

D — Misses that the harness, shot count and aggregation are part of what produced the number, so without them there is no defined score to validate.

10. Publishing weights people can trust — C

A — A loss curve describes the run, not the capability; a converged model can still be no better than what it started from.

B — Parameter count constrains capacity but says nothing about whether the tuning improved the task.

D — Useful context, but a general benchmark cannot show what a task-specific tune achieved on that task.

11. The Families, and What Each Is Actually For — A

B — Treats a permissive licence and a high score as sufficient, ignoring that the flagship cannot fit on the target machine at all.

C — Reads benchmark position as evidence about long-tail multilingual behaviour, which is precisely where synthetic-data training is weakest.

D — Assumes 8B parameters means 8GB; at bf16 the weights alone are about 16GB before any context.

12. Hugging Face as Infrastructure — A

B — Confuses integrity (the file is what it was) with safety (what the file does when opened).

C — A true property of the format, but it answers a security question with a speed argument.

D — Confuses the container format with the dtype inside it; either format can hold any precision.

13. Regional and specialist models — C

A — A real confound worth controlling, but it would degrade the large model on both tasks rather than only on Tamil.

B — Confuses task difficulty with language coverage; the same 7B would likely lose at English summarisation too.

D — Public Indic benchmarks are thin, but this result came from the team's own examples, which is exactly the evidence that does not depend on public benchmarks.

14. Models that write code — C

A — Latency affects speed, not output shape; a slow completion model still returns a code fragment.

B — Close, but the template is a symptom: an instruct model answers conversationally because that is what it was tuned to do, template or not.

D — Temperature changes which tokens are chosen, not whether the model believes it is in a conversation.

15. Embeddings you can host yourself — A

B — Higher dimensions do change distance behaviour in general, but retrieval models are trained at their dimension and do not degrade for that reason.

C — A real bug worth checking, but it would affect thresholds rather than ranking, since cosine ranking is unchanged by a consistent scale.

D — Would apply to the monolingual variant only, and would not explain a drop against MiniLM, which is also English-centric.

16. Rerankers, and why retrieval is two stages — C

A — Worth fixing eventually, but 4% missing candidates cannot explain a 30% top-5 failure rate.

B — A better embedding model mostly improves candidate recall, which is already the healthy number here.

D — Bad chunking usually shows up as poor recall at 100, because a split answer is hard to retrieve at all.

17. Speech, in both directions — D

A — Encoding quality affects accuracy on real speech but does not cause fabricated text in genuine silence.

B — Lowering temperature reduces variety but the model still emits its most likely completion for a segment it should not have transcribed at all.

C — Mislocates the cause: the silences are inside the recording, not appended to it.

18. Image and video weights, and their licence traps — B

A — Model licences routinely restrict use as well as redistribution, which is precisely how a non-commercial licence works.

C — Confuses the separate question of whether output is copyrightable with whether the licence permits the use.

D — A real and unsettled issue, but it is not what makes this specific use a licence breach.

19. What a model of each size can honestly do — B

A — Plausible only if the workflow is long in tokens; five short steps fit easily, and the arithmetic already explains the result.

C — Assumes step successes combine additively rather than multiplicatively, which is the error the arithmetic exposes.

D — A larger model raises per-step accuracy but the compounding remains; at 96% per step a five-step chain is still only 82%.

20. Renting open weights by the token — C

A — Possible in principle, but published open weights are versioned repositories and providers pin them; routing is the far more common cause.

B — Worth checking, but a step change in one week with the same document pipeline points at the serving side.

D — Confuses speed with quality; the same weights and sampler produce the same distribution regardless of how long it takes.

21. Base, Instruct, Reasoning — A

B — Blames numerical precision for what is a mismatch between the model's training objective and the task's shape.

C — Assumes capability is monotone across variants and that any regression must be a prompt defect.

D — Inverts the role of sampling: determinism is what an extraction task wants, not what is suppressing it.

22. Every number in config.json — B

A — Reverses the relationship: at a fixed total size, embedding parameters are taken away from the layers, not added to them.

C — Treats the vocabulary as tokenizer-only, but the embedding and output matrices are both sized by it and live in the weights.

D — KV cache depends on layers, key-value heads and head dimension; vocabulary size does not appear in it.

23. Why the same sentence costs different amounts — C

A — Reply length is a behavioural property of the tuning; the saving here comes from how the same text is segmented, not from shorter answers.

B — Batching efficiency is set by memory and sequence length, and providers price per token regardless.

D — Confuses text segmentation with weight storage; vocabulary size increases the embedding matrices rather than compressing anything.

24. The chat template is a contract — B

A — Would cause generation to run to the token limit mid-sentence, not to produce a well-formed new user turn.

C — Patches a formatting fault with prompt text; the model is completing a document because it was never told the assistant turn had begun.

D — Sampling changes word choice, not whether the model believes the transcript has ended.

25. Attention variants, and the cache that eats your memory — C

A — More key-value heads add memory traffic rather than throughput; if anything the larger cache slows decoding.

B — Counts only the weights; the cache is a separate allocation that scales with key-value heads and grows per user.

D — Reverses the relationship: a larger cache per token means fewer tokens fit in the same memory, and context capacity is unrelated to head count.

26. What a 128k context window really means — C

A — Would predict failure on the 120k needle test too, which passed.

B — Eviction is a serving configuration, and it would degrade the single-needle result at 120k first.

D — Tokenisation is uniform across a document and does not explain a task-difficulty gap at the same length.

27. Mixture of experts, and the memory it still needs — C

A — Describes the compute side correctly while ignoring that all experts must be resident before any of them can run.

B — Assumes selective loading; the router can pick any expert for the next token, so partial residency means constant reloading.

D — MoE models are in fact somewhat more sensitive to aggressive quantisation, and in any case the total size is what has to fit.

28. Models that can see — C

A — Context is not the binding constraint here; the model is misreading layout, not running out of space.

B — Greedy decoding removes randomness but the model's most likely reading of an ambiguous layout is still wrong.

D — Worth testing, but layout parsing is a known category weakness across sizes and the confident-wrong behaviour persists.

29. Formats, and what conversion costs you — B

A — Worth checking, but a Q4 gap on ordinary tasks is usually small; a large gap points at something structural.

C — A short default context truncates long inputs but does not degrade short-prompt quality.

D — Arithmetic order differs slightly between backends, but the same weights and sampler give equivalent quality on either.

30. Why it will not stop talking — C

A — Context overflow produces degraded or truncated text, not a correctly emitted special token rendered as visible characters.

B — The model emitted the right token at the right moment; the fault is entirely on the decoding and stop side.

D — A template fault usually shows as the model writing a new user turn; here the model signalled the end properly and the server ignored it.

31. Choosing a Model for the Job — A

B — A plausible-sounding load-time myth; quantized formats are read directly and never expand to full precision in RAM.

C — Confuses slow with unsupported; CPU inference works at any size, it just gets impractical.

D — Blames a fixed-size component; embeddings depend on vocabulary size, not on how many tokens you feed in.

32. What quantisation actually does — B

A — Those files are a few megabytes; they cannot account for a gigabyte.

C — Embeddings are often kept at higher precision than the average, but they are quantised too and are not the main cause.

D — The cache is allocated at runtime from your memory and is not stored in the model file at all.

33. Choosing a bit width you can defend — C

A — The size sensitivity is real below roughly 3B; an 8B quantises well on English, which is why the language is the distinguishing factor here.

B — Conversion can damage templates and stop tokens, but the vocabulary is copied wholesale and a missing token set would break generation entirely.

D — Perplexity is genuinely a weak proxy here, but the corpus is not the core issue: the damage is unevenly distributed across capabilities.

34. llama.cpp, end to end — B

A — Assumes all-or-nothing placement, which is exactly what layer-wise offload avoids.

C — Confuses offload with quantisation; -ngl moves layers, it does not change their precision.

D — Per-token swapping over the bus would be far slower than simply running those layers on the CPU, which is why it is not what happens.

35. Ollama and the friendly path — B

A — Quantisation degrades quality gradually; it does not make the model behave as though the earlier text was absent.

C — A wrong template garbles roles and stop behaviour rather than selectively discarding the first part of a single message.

D — Worth ruling out, but it would not produce this pattern with no system prompt set, which is the default here.

36. Predicting speed before you download — C

A — Lower precision reduces bytes read, which is already accounted for in the 8 GB figure; it does not remove the bandwidth limit.

B — Confuses prefill with decode; a long prompt raises prompt-processing throughput and slightly slows decoding.

D — The cache the operating system keeps is in the same memory; CPU caches are megabytes and cannot hold an 8 GB model.

37. Serving many users at once — D

A — vLLM does not quantise a model unless asked, and quantisation would affect decoding rather than the shared prompt.

B — Tensor parallelism splits a model too large for one card and adds communication cost; it does not deduplicate repeated prompts.

C — Continuous batching does raise throughput, but it does not stop the identical 2,000-token prefix being recomputed for every request.

38. Running all of this without money — B

A — Fitting in memory is a precondition for running at all; it has nothing to do with surviving a disconnect.

C — Neither preserves a running session; both discard the local disk when the session ends.

D — Gradient checkpointing trades compute for memory during backpropagation and has nothing to do with saving progress.

39. Rent, or own: the arithmetic — B

A — Latency to a provider is usually tens of milliseconds against generation times measured in seconds, so it rarely justifies a cost inversion.

C — True as a trend and worth noting, but it is not the error in the arithmetic as presented.

D — The same weights on adequate hardware give the same quality; the difference is in batching efficiency and utilisation, not capability.

40. What actually runs on a phone — C

A — Memory pressure causes termination or reload stalls rather than a steady halving of throughput.

B — A real distinction, but the figure quoted is already generation speed, and the reported slowdown develops over time rather than being constant.

D — Power modes reduce clocks, not model precision; the weights on disk do not change.

41. The sampler, and the defaults that ship broken — A

B — Truncation cuts output off entirely rather than corrupting punctuation while continuing to produce fields.

C — High temperature would corrupt output from the start rather than degrade progressively with each repeated symbol.

D — A plausible-sounding capacity story, but it would not explain why short objects are perfect and the degradation tracks repetition.

42. Making the output valid by construction — B

A — Masking removes illegal tokens rather than changing the temperature applied to the legal ones.

C — Compilation is a one-off cost and would produce missing responses, not lower-quality complete ones.

D — Structured output is usually shorter than prose, and truncation would show as parse failures, which have gone to zero.

43. What changes when nothing is hidden — C

A — System-prompt adherence is exactly what weakens at small sizes, so this moves the instruction somewhere it binds less.

B — Temperature affects token choice within the model's distribution; the concept is already activated by the negation.

D — A reasonable safety net, but it patches the symptom and leaves the model's reasoning shaped around the forbidden concept.

44. Examples, and how to choose them — B

A — An assertion about priors competes weakly against the demonstrated distribution, particularly at small sizes.

C — More examples in the same proportion preserve the imbalance while costing more context.

D — Loses the format and label-space benefit that examples provide, which is usually a larger effect than the imbalance.

45. Tool calling, and what breaks at 8B — C

A — Possible, but an untrained model usually produces inconsistent or prose-wrapped output rather than a well-formed call in the expected shape.

B — If the model emitted the correct family-specific format, the definitions almost certainly did reach it.

D — Constrained decoding governs output validity; the output here is already valid and the problem is that nothing extracted it.

46. Agent loops on models that are not clever — A

B — Treats per-step accuracy as the end-to-end rate; five steps at 0.97 is 0.86, not 0.97.

C — Self-correction happens occasionally but is not reliable at any size, and assuming it is the reason 60% was observed with a 92% step rate.

D — Inverts the arithmetic: 0.92 to the fifth power is about 0.66, which is close to the observed rate.

47. What these models really do in your language — B

A — Possible, but contamination would inflate accuracy rather than produce fluent-but-odd prose that users can read and object to.

C — A genuine effect worth ruling out, but it would tend to produce errors rather than a consistent register mismatch.

D — High fertility raises cost and shortens effective context; it does not by itself produce the wrong register.

48. Refusals, and the uncensored question — C

A — Reduces the rate but provides no guarantee, since the behaviour is a learned default rather than an enforced rule.

B — Makes refusal more likely and remains a behavioural default that prompting can sometimes route around.

D — Placement genuinely helps adherence, but a prompt is an instruction to the same model whose output you are trying to bound.

49. Guardrails as a separate layer — C

A — Size affects accuracy on the inputs it sees; it cannot help with a stage the guardrail never inspects.

B — A real category of failure, but the premise states the inputs were genuinely benign, so nothing needed evading.

D — Quantisation can shift refusal behaviour slightly, but the structural gap here is that nothing examined the output at all.

50. When the model does not know — B

A — Users also experience the confident wrong answers on unanswerable questions, which A produces 90% of the time.

C — Assumes the abstention is untargeted, but B's answerable accuracy of 82% is measured separately and shows the cost is bounded.

D — Token probabilities in free-form output are a weak and poorly calibrated signal, which is why schema-level abstention is used instead.

51. When not to fine-tune — B

A — Volume is not the binding issue; 2,000 examples is ample for behavioural tuning, and more would not fix the mismatch.

C — Forgetting is a real risk to measure, but it is a side effect rather than the reason this plan is the wrong tool.

D — A larger model still will not contain their private specifications, so the same gap persists at greater cost.

52. What a LoRA actually is — B

A — Overfitting produces worse generalisation, not a smaller departure from the base on training-style inputs.

C — Extra parameters do train; the scaling factor rather than data volume explains a systematically weaker effect.

D — Module choice matters independently, but it was unchanged here, so it cannot explain a change in behaviour.

53. Fitting a fine-tune in 16 GB — C

A — Saves well under a gigabyte of gradient and optimiser state while removing capacity, so it is the least effective change available.

B — Moves in the wrong direction: an 8-bit base doubles the frozen weight memory.

D — Accumulation helps only if the per-device batch size falls with it; raising the step count alone changes nothing about peak memory.

54. Five hundred examples, done properly — B

A — Improving the guidelines is the right response, but the current data already carries the disagreement into the model.

C — Thirty items gives a rough but actionable figure, and measuring the model against inconsistent labels will not reveal the cause.

D — Preference tuning also requires consistent judgements; the ambiguity has to be resolved in the guidelines either way.

55. A real run, on free hardware — C

A — Noise is worth checking, but loss and task accuracy measure different things, so agreement between them was never guaranteed.

B — A rate that damages the model raises loss rather than lowering it.

D — Possible in the next epoch, but it does not explain lower loss with worse accuracy at the current checkpoint.

56. Teaching it which answer is better — C

A — Inverts the parameter: a higher beta constrains drift, and a lower one permits it.

B — Plausible on the surface, but the systematic length increase points to the judge's preference rather than a coverage gap.

D — A low rate produces smaller changes overall rather than selectively teaching surface features.

57. Distilling a larger model into a smaller one — B

A — Agreement is not a capability ratio, and disagreements concentrate on the hard cases where capability differences show.

C — Higher agreement would mean closer imitation, which is not the same as better task performance.

D — A student trained to imitate has no mechanism for exceeding its teacher on the distilled behaviour.

58. Measuring what you broke — B

A — Fine-tuning does not alter the tokenizer unless vocabulary was explicitly extended, which is a separate procedure.

C — Selective application is a real fix, but the degradation exists because the tune damaged the capability, not because the adapter is misrouted.

D — A genuine effect from an earlier lesson, but it would have been present before the fine-tune too.

59. Merging models, and what it really does — C

A — Licences do not combine by vote; every parent's obligations attach to a derivative that contains its weights.

B — Overstates the position: the merge can generally be distributed under the most restrictive parent's terms, not under none.

D — Averaging weights is derivation in the most literal sense; the parents' parameters are arithmetically present in the result.

60. When the model does not have your language at all — C

A — Templates are not used in raw-text pretraining by design; applying one would not have taught instruction following.

B — Would corrupt the language output too, whereas the model writes fluently.

D — Names a plausible mechanism for damage, but the loss of instruction following is the expected outcome of the objective, not of the rate.

61. Shipping the adapter — B

A — Merged weights convert normally; that is one of the reasons people merge.

C — The base remains downloadable and comparable; what is gone is the option to serve it alongside the tune from one deployment.

D — A new adapter can still be trained against the original base; the merge does not prevent future tuning.

62. What Benchmarks Actually Measure — A

B — Treats sampling error as the only source of uncertainty, ignoring harness differences and label errors in the benchmark itself.

C — Assumes a shared benchmark name implies a shared evaluation protocol, which it does not.

D — Over-corrects into blanket cynicism; the useful move is asking about the protocol, not refusing all vendor data.

63. Reading a model card for what it does not say — B

A — Contamination is a real limit on MMLU, but it does not explain why these two particular numbers cannot be compared with each other.

C — Recency is worth checking but the deeper issue is that the two figures were never produced under one configuration.

D — Five points is larger than typical run-to-run noise; the problem is that these runs were not comparable in the first place.

64. The leaderboards that still carry information — B

A — Treats overlapping intervals as if the point estimates settled the question, which is what the intervals exist to prevent.

C — Preference is influenced by style, but the ranking does not decompose into style and substance, and the gap is not established at all.

D — More votes would narrow the intervals, but until they do the ordering is unresolved rather than correct-but-imprecise.

65. Running the standard harness yourself — C

A — Quantisation does cost something, but rarely four points on MMLU, and the team would usually know which file they loaded.

B — Sampling adds variance around the mean rather than a consistent four-point deficit.

D — Contamination is a property of the training data, so it is identical in both copies of the same weights.

66. Proving what a quantisation cost — B

A — Perplexity averages log-likelihood over all tokens and can hide large disagreement about which token wins.

C — Aggressive but valid quantisations routinely reach that range; it is a quality cost, not corruption.

D — The measurement compares distributions rather than samples, so stochastic sampling does not enter into it.

67. Evaluating an embedding model on your corpus — B

A — Worth standardising, but cosine ranking is unaffected by a consistent metric choice across a single model's index.

C — A general concern about set quality, but it would depress all three models equally rather than one.

D — Domain fit is a real factor, but it does not explain a result that a configuration check may fully account for.

68. Measuring latency properly — B

A — Aggregate throughput is a server-capacity number and does not describe what one user waits for.

C — Worth checking, but a tail regression affects a small share of requests rather than making the assistant feel sluggish generally.

D — More output would raise total latency, which the premise says is unchanged.

69. Cost per task, not cost per token — C

A — Inverts the arithmetic: 400 tokens at 2.00 is 800 microdollars against 280 for A, so the price difference dominates here.

B — Output is identical, but input differs by a factor of nearly three in cost and is the larger term for both.

D — A ten-times price gap is closed by a 3.5-times length gap plus retries or caching, so the conclusion is not robust.

70. Swapping the model under a live product — B

A — Whether six points is noise depends entirely on set size, which is a separate question from the comparison being unfair.

C — Public benchmarks are less relevant to the task, and moving the comparison away from their own data weakens it rather than fixing it.

D — Fine-tuning is a much larger commitment than the hour of prompt tuning that would make the comparison fair.

71. Evaluate a Model on Your Task in an Afternoon — A

B — Treats a point estimate as a measurement, ignoring how wide the interval is at forty examples.

C — Uses a plausible rule of thumb to skip the statistics; the gap may not exist to close.

D — Mines variance instead of reducing it, which is exactly the aggregation trick that inflates published benchmarks.

Module test — 1. What open actually means

1. A

A licence travels with the weights, not with the data that shaped them. The distill is a Llama model that was trained on R1's traces, so the parameters on disk are Meta's and Meta's terms apply — the naming rule, the attribution line and the user cap included. Follow the base_model field down the chain until you reach a repository that has none; that is where the obligations start.

2. D

Each axis buys one concrete capability, and contamination checking is bought by data disclosure alone. If the corpus is published you can grep it and answer the question; if it is not, every answer is a guess. That is why a release scoring two on weights and zero on data can still be useless for a regulator or a security review, and why the rubric is more useful than arguing about the word open.

3. C

Every opt-out mechanism is prospective. Snapshots already taken are published and mirrored, and a model that has already learned from them cannot have that learning removed short of retraining. The line in robots.txt is worth adding, but it protects work published from today onwards, not the archive that was on the open web in 2019.

4. B

Safetensors closes one execution path: the weights file is a header plus raw bytes and loading it runs nothing. The flag opens a different one by design — it downloads the repository's own modelling and configuration Python and imports it. The remedy is unglamorous: read the file, which is usually a few hundred lines of tensor operations, and pin the revision so it cannot change under you afterwards.

5. A

Downloads track whichever quantised repository got linked from a popular post, and popularity in this ecosystem moves in days. A limitations section costs the uploader effort and admits something unflattering, which is exactly why it carries information: nobody can write one without having run the model on things it failed. It is the strongest single signal on a stranger's model card.

6. D

The exemption is partial and two duties sit outside it: a copyright policy that respects machine-readable rights reservations, and a published summary of the training content following the AI Office template. Both survive an open release. The exemption also stops entirely above the systemic-risk compute threshold, which is a frontier-scale number that ordinary fine-tuning is nowhere near.

Module test — 2. The map, and how to read it

1. D

A general model carries two disadvantages in another language and both are measurable. It has seen less of it, and its tokenizer chops it into more pieces. A tokenizer trained with the language in it can cut tokens per word from six or seven down to two or three, which means the same document fits, each request costs less and generation is faster — before any argument about quality. Compare token counts as part of the comparison, not after it.

2. C

Several embedding families are trained asymmetrically: the query is embedded with one prefix and the passage with another. Embedding both sides identically raises no error and produces vectors that are simply worse aligned, so retrieval quietly degrades. Read the card for the exact strings, because they differ per family, and treat any silent quality loss in retrieval as a configuration question before a model question.

3. B

A reranker only reorders what it is given; it cannot produce a document the first stage missed. Recall at 100 of 0.52 means the answer is absent from the candidate set for nearly half the queries, and no reranker will recover those. This is why the two stages are measured separately — recall at 100 for the retriever, precision in the top five after reranking — and why teams reporting one end-to-end number spend weeks tuning the wrong stage.

4. A

Whisper invents text on silence, and the invented text is usually a phrase that was common in its training subtitles, such as a thank-you or a channel sign-off. It is not a sampling artefact and a bigger model does not fix it. Voice activity detection finds the speech regions and the transcriber only ever sees those, which is the difference between a transcript you can trust and one that fabricates sentences in the gaps.

5. D

One family, three variants, three different answers. The schnell variant is Apache 2.0 and commercially usable; dev is non-commercial and its terms reach the outputs, so generating with it and selling the picture is not a workaround; pro was never released. This is the licence trap that catches somebody every week, and the habit that avoids it is to check the variant rather than the family before anything reaches a customer.

6. C

Two providers running the same repository are not running the same system. One serves bf16, another fp8, another int4; context caps differ; default temperature and repetition penalty differ and are applied when you do not specify. Cheaper prices usually mean more aggressive serving, and it is rarely on the pricing page. Pin the provider and the quantisation, record both beside the model revision, and re-run the evaluation after any routing change.

Module test — 3. Inside the files

1. C

Per layer the four attention projections come to about 42 million parameters, while the three gated feed-forward matrices at 4096 by 14336 come to about 176 million. That is four fifths of the layer in the part nobody writes about. Knowing it changes what you look at in a configuration file: the intermediate size to hidden size ratio tells you about memory, and the key-value head count tells you about serving.

2. B

Cache bytes per token are two times layers times key-value heads times head dimension times bytes per element. Only the key-value head count changes between these two, so the cache scales by the ratio: 32 heads to 8 is a factor of four. At 128 kilobytes per token a ten-user server at 8k context needs about ten gigabytes of cache on top of the weights, so this one field decides how many people one GPU will hold.

3. A

Without the generation prompt the formatted string stops after the user's turn, and the model has never been told it is the assistant's turn. It then does what its training says a document like that does next: it writes another user message. Print the exact string you are sending, compare it against the template output for the same messages, and check it ends with the assistant header. Thirty seconds, and it rules out the likeliest cause.

4. D

The template already inserts the beginning-of-sequence token. Tokenising its output with special tokens enabled adds another, and two of them at the start measurably degrades output on some models while raising no error at all. Either tokenise inside the template call and let it do both jobs, or disable special tokens when tokenising a string the template has already built.

5. C

Retrieving one distinctive sentence is close to the easiest thing a long context can be asked to do, which is why the charts are green. Harder tasks — several needles, tracking a variable, aggregating facts, or answering that the answer is absent — show effective lengths often a quarter or less of the advertised number. Measure your own: three questions at different depths on five real documents, then halve the length and repeat.

6. B

Modern models often carry several terminal ids — an end-of-text token and one or two end-of-turn variants. If a runtime honours only the first, the model emits the right token at the right moment, the runtime does not recognise it, and generation continues as a document, which is why the next thing it writes is a new user turn. Log the raw token ids where you wanted it to stop, and pass the stop strings explicitly.

Module test — 4. Making it run on the machine you have

1. B

Quantisation is block-wise: a block of 32 weights stores 32 four-bit indices plus a scale, and often a minimum, in 16 bits each. That is 160 bits for 32 weights, or five bits per weight rather than four. Real formats land between 4.5 and 5.5 bits once the scales are counted, which is exactly the gap between the number in the name and the size of the file.

2. A

Spend memory on parameters before precision, down to about four bits. Quantisation works because many weights each contribute a little, so rounding errors average out, and a larger model has more of that redundancy to spare. The rule inverts below about three billion parameters and below four bits, where there is less redundancy and the curve steepens — so a very small model deserves Q6 or Q8.

3. D

To produce one token the machine reads every active weight from memory once, so tokens per second is bounded by bandwidth divided by the size of those weights. Fifty divided by five is ten, and real numbers land at sixty to eighty per cent of the ceiling. Sixty tokens per second from a five-gigabyte file implies about three hundred gigabytes per second, and the right question is which machine provides it.

4. C

The two phases are limited by different resources. Prefill pushes every prompt token through the weights together, so the weights are read once for the batch and the cost is arithmetic, which processors have in abundance. Decode reads the weights again for each token, so it is bandwidth-bound. Hence the practical consequence: a long prompt with a short answer is fast, and a short prompt with a long answer is slow.

5. B

Friendly wrappers choose several things for you, and the context default is the one that bites hardest. A prompt longer than the configured window has its beginning silently dropped, which produces exactly this symptom and gets reported as the model being poor. Set the context length explicitly, confirm what you actually have, and record it alongside the quantisation and the sampler with any result you intend to act on.

6. A

Breakeven against a rented endpoint already sits in the hundreds of millions of output tokens a month, and that figure assumes the machine is busy. Real traffic is bursty — quiet at night, peaked in the afternoon — and a rented endpoint absorbs that for free because somebody else's traffic fills the gaps. A card at five per cent utilisation costs twenty times its sticker price per token delivered.

Module test — 5. Getting good output

1. A

JSON is made of repeated tokens: the quote, the colon, the comma, the braces. A repetition penalty divides the logits of tokens that have already appeared, so by the fifth field the model is being actively discouraged from producing correct syntax. Set it to one for any structured output. If the model genuinely loops, the cause is the prompt or a missing stop token, and the penalty is damaging the format to treat a symptom.

2. D

Constrained decoding guarantees syntax and guarantees nothing about truth. A model forced to emit an amount will emit one whether or not the document contains it, so the trade you made was a visible parse failure for a silent wrong value. Always give the model somewhere to go: a nullable field, or an explicit unknown in the enum, and say in the prompt that it is a correct answer.

3. C

Telling a small model not to mention something puts the concept in its context, and suppression is a logical operation small models are weak at. Naming what to discuss instead removes the concept rather than forbidding it. Position matters for the same reason: attention to recent tokens is stronger, so the rule that must hold belongs immediately before the model generates, not buried at the top.

4. B

Examples mainly teach three things: what the inputs look like, what the label space is, and exactly what shape the output takes. The mapping is carried less than intuition suggests. So the highest-return properties are byte-identical formatting across examples and balanced coverage of every label — a set with four billing cases and one fault case will over-predict billing — rather than subtlety in the individual answers.

5. A

A hallucinated observation is dangerous because everything downstream looks fine. Asking the model to attest to its own behaviour puts the check inside the thing being checked. The fix belongs in the loop: your code executes actions and your code supplies observations, and any observation text the model wrote itself is ignored. The same principle covers the state, the deduplication and the step cap.

6. D

A fabricated quotation fails a string search, which makes it a hard check rather than a soft one, and it is the most effective technique after retrieval itself. Token probabilities are only weakly informative on free-form text — preference tuning rewards confident-sounding answers, so a model can assign very high probability to an invented citation — and self-rated confidence inherits the same problem.

Module test — 6. Adapting a model to your work

1. D

Fine-tuning reliably changes form — format, tone, register, which of several acceptable answers is preferred — and changes knowledge badly. The model learns the surface of a statement, generalises it unpredictably, contradicts it when the question is phrased differently, and forgets other things while doing so. Retrieval is better on every axis: cheaper, updatable, citable, and it degrades nothing.

2. C

The adapter contributes alpha divided by rank times B times A. Holding alpha at 32 while moving rank from 16 to 64 quarters the effective strength of the update, so a run with more capacity does less. The convention that alpha is twice the rank exists precisely so the two move together. Print the trainable parameter count every run; a number far from what you expected means the settings are wrong before the hours are spent.

3. B

QLoRA has already collapsed weights, gradients and optimiser state; what is left is activations, and activation memory scales with batch size times sequence length. Gradient checkpointing recomputes most activations instead of storing them, trading about thirty per cent more compute for a large memory saving, and a sequence length matched to your real data stops you paying for padding. Reducing rank is last, because it is the only change that reduces what the run can learn.

4. A

If two careful people agree eighty per cent of the time, your labels encode eighty per cent agreement and no amount of tuning gets past your own disagreement. The measurement takes an hour and reframes the project: it tells you what a good score looks like, it often reveals that the task definition rather than the model is the problem, and it stops a team chasing ninety-five per cent on a task nobody can define that precisely.

5. D

Fine-tuning moves the weights that held everything else, so a tune can gain nine points on the task and lose fifteen on general instruction following, other languages, other formats and refusal behaviour. Fifty items covering those four areas plus the task at longer lengths, recorded against the base model first, is an hour of work and turns a headline number into a decision. Report both numbers with every tune.

6. C

Fine-tunes of a shared base have not travelled far from the base or from each other, so they remain in a connected region where the loss is low and points between them are also reasonable models. Two models with different bases occupy unrelated parts of weight space and the midpoint means nothing. This is also why task arithmetic works: you are adding and subtracting the deltas, not the models.

Module test — 7. Judging models and claims

1. C

The two sets are matched for style and difficulty, so difficulty is held constant and the harness is the same for both. What differs is that one set was on the public web long enough to be crawled into training data and the other was not. A large drop is the portion of the original score that was recitation. A flat result is evidence that the model was solving the problems.

2. B

A benchmark number is produced by a pipeline — model, prompt format, shot count, decoding settings, answer extraction, harness version — and changing any part of it moves the number by several points. Without those facts the figure cannot be reproduced even in principle, and it cannot be compared with anybody else's. That does not mean the model is bad; it means the table is a claim and you will have to measure it yourself.

3. A

Perplexity is a screening test: it is cheap, it correlates with quality, and it is not sensitive enough to settle this. Divergence from the full-precision reference asks a sharper question — how far the output distribution moved — and the fraction of tokens where the top choice differs is directly interpretable. Under about one or two per cent is quiet; above about five, expect visible behavioural differences. Report the tail as well as the mean.

4. D

Chunking decides what a vector has to represent, and a chunk that splits an answer in half or buries it among four other topics costs recall that no model recovers. It is also the variable almost nobody tests. Nine configurations take a few minutes each to index with a small model on a CPU, and the best chunk size depends on your documents rather than on general advice — which is the point of running it on your own corpus.

5. C

Total time is prefill plus decode, and the two feel completely different. Two hundred milliseconds to first token followed by a steady stream reads as fast; three seconds of nothing followed by an instant block reads as broken, even when the totals match. Report time to first token and per-token latency separately, at p95 and at real concurrency, and remember that streaming makes nothing faster while changing which number the user experiences.

6. B

Cost per task is input tokens times input price plus output tokens times output price, all multiplied by one plus the retry rate. Examples carried on every request, retries from invalid output, and a reasoning model's thinking tokens billed at the output rate each invert the ranking of a price list routinely. Log input tokens, output tokens, attempts and the model per request, and the cost becomes a query rather than an argument.

The Open Model Ecosystem — final examination

1. D ***1. What open actually means***

There are four things weights alone cannot give you — auditing the data, reproducing the model, removing something it learned, and explaining a behaviour by pointing at data. There is one they do give you, and it is the durable benefit: permanence. A model that is withdrawn from a hub, deprecated, repriced or declared off-policy still runs from your own storage. That is the whole reason open weights matter, and it is enough.

2. A ***1. What open actually means***

The definition asks for the parameters, the complete training and inference code, and data information detailed enough that a skilled person could build a substantially equivalent system. Parameters are published routinely and inference code is ubiquitous; the data information is the part almost nobody releases. That is why three labels are worth keeping separate — open weights, openly licensed weights, and fully open — and why only the third makes the phrase open source honest without qualification.

3. B ***1. What open actually means***

Meta's terms require a derived model's name to begin with Llama, alongside a prominent built-with notice and an attribution line. Separately, a fine-tune inherits the base model's terms, so relicensing it as Apache 2.0 does not make it Apache 2.0 — it makes the card wrong. Both are checkable in ten minutes and both are found by whoever runs diligence on the company later.

4. C ***1. What open actually means***

Releases are strategy rather than principle, and the strategy has to close somewhere. A lab that sells advertising, cloud capacity or reputation can commoditise the model layer indefinitely. A company whose only product is the model cannot, and the pattern that follows is familiar: a paid tier, a changed licence at the next release, or an acquisition with different lawyers. Keep the weights you depend on, and record the terms you accepted with the date.

5. D ***1. What open actually means***

No filtering step is neutral. A naive profanity filter treats reclaimed language as profanity, quality classification is a taste judgement wearing a statistical hat, and language identification is least accurate exactly where a language's data is thinnest, so low-resource languages quietly disappear. None of this is disclosed in enough detail to reproduce for most commercial releases, which is itself the point of the data axis on the openness rubric.

6. A ***1. What open actually means***

The three questions are independent and most confused arguments are two people answering different ones. Whether training copies lawfully is one. Whether a particular generation reproduces protected expression is another, and it is the one you control. Whether the generation can itself be owned is a third, answered by human-authorship rules that vary by jurisdiction, and it has nothing to do with the first two.

7. B ***1. What open actually means***

Hub-side scanning is a filter, not a proof: it recognises known-bad pickle patterns in files. A backdoor planted in weights or an adapter is a behaviour, and behaviours have no signature to match. The benchmarks pass because the benchmarks do not contain the trigger. The defences are procedural — prefer adapters from the organisation that made the base or from an attributable team, evaluate on your own set, and keep such a model out of any path that writes to production.

8. C ***1. What open actually means***

Your score on its own says nothing about what your work did; the reader has to assume the base was already there. Running the base under the same harness, the same shot count and the same decoding settings, and printing the two numbers side by side, is the only comparison that isolates the tune. Numbers collected from three different papers are a collage, not a comparison, however impressive the table looks.

9. D ***2. The map, and how to read it***

Any ranking of these families has a shelf life of about a quarter. What is durable is the shape: what sizes exist, what terms they ship under, what tooling assumes them, what languages they cover, and the house style. Narrow on those, because they are still true next year, then test the two or three survivors on your own examples — family reputation tells you nothing about your task.

10. A ***2. The map, and how to read it***

A model repository is a git repository with large-file storage, and the main branch moves: people push new quantisations, fix a broken chat template, or change a configuration quietly. Pinning the commit hash — in the download command and in the loading call — is the difference between a build you can rerun in six months and one you cannot. Record that hash in the model inventory next to the licence.

11. B ***2. The map, and how to read it***

Code models split into two tools that people constantly confuse. Completion models are trained with fill-in-the-middle and take special prefix, suffix and middle tokens — no conversation at all. Instruct models expect a chat template and answer questions. Asking an instruct model for inline completion produces exactly this: prose where four tokens were wanted. Every serious code family ships both, and the base repository is usually the one with fill-in-the-middle.

12. C ***2. The map, and how to read it***

Truncation in embedding models is silent. The vector represents the first five hundred and twelve tokens and the rest of the chunk simply has no representation, so a query answered by the second half will never retrieve it and nothing anywhere reports a problem. Either chunk to fit the model, or choose a long-context embedding model deliberately. The same silence applies to missing query and passage prefixes.

13. D ***2. The map, and how to read it***

Dense retrieval compresses a document before the query exists, and exact strings — product codes, error numbers, surnames, unusual acronyms — are precisely what compression destroys. Keyword scoring has the opposite profile: it finds exact tokens and misses paraphrases. Running both and merging the ranked lists by reciprocal rank fusion covers two different failure modes with two different mechanisms, which is why most production systems converge on it.

14. A ***2. The map, and how to read it***

Piper is MIT, runs in real time on a Raspberry Pi and covers dozens of languages. It does not sound like the newest demos, and for announcements, accessibility and interactive voice response that is the right trade. The quality ladder in synthesis costs compute in a way recognition does not — a good neural voice is several times more expensive per second of audio than transcribing the same second — and cloning a real person's voice needs their recorded consent.

15. B ***2. The map, and how to read it***

Multi-step reliability compounds, so a model that is fine on single questions falls apart when three jobs share one call. A model failing at classify-and-extract-and-summarise usually succeeds at each separately, and three calls to a small model are cheaper and better than one call to a large one — and each is separately testable. Narrowing the task is the first move; giving it the information, constraining the output and showing examples come next.

16. C ***2. The map, and how to read it***

An aggregator passes your request to somebody else, so the effective policy is set by every link, not by the page you read when you signed up. Ask three questions of any route before sending real data: whether prompts and completions are retained and for how long, whether they are used for training and whether opting out is the default, and where the request physically executes. Where a no-logging setting exists, understand that it constrains which providers can serve you.

17. D ***3. Inside the files***

A reasoning model buys accuracy on multi-step problems by spending three to twenty times the output tokens and seconds of latency. Most production tasks are not multi-step, and on a short structured task the deliberation is pure cost and an occasional source of error. Base, instruct and reasoning are not a quality ladder; they are three different contracts about what the model does with your prompt.

18. A ***3. Inside the files***

The embedding table is vocabulary size times hidden size, and unless the output head is tied it appears twice. At 256,000 by 2048 that is over half a billion parameters in a one-billion model, before a single layer exists. This is the trade a large multilingual vocabulary makes, and it is why very small models often carry small vocabularies and therefore tokenise other languages badly.

19. B ***3. Inside the files***

Price is per token, context length is in tokens and speed is tokens per second, so fertility multiplies straight into all three. A thirty-two-thousand-token window holding twenty-four thousand English words might hold six thousand Hindi words on a poor tokenizer, and a reply of the same length takes three times as long. Measure it on a thousand of your own documents; teams routinely find a thirty per cent cost difference between models they were choosing on quality alone.

20. C ***3. Inside the files***

The template ships with the model in its tokenizer configuration and is the authoritative statement of its format. Some families have no system role at all, so a system message is either rejected or silently folded into the first user turn; others accept one but were tuned with almost none and follow it weakly. If the word system does not appear in the Jinja source, prepend your instructions to the first user message and expect them to bind about as strongly.

21. D ***3. Inside the files***

Compute scales with active parameters and memory scales with the total, because the router can send the next token to any expert and so every expert must be resident. Thirty billion at four bits is about eighteen gigabytes. On a thirty-two-gigabyte workstation this design is the best value in the ecosystem; on an eight-gigabyte laptop a four-billion dense model wins, because the alternative to fitting is not running.

22. A ***3. Inside the files***

A vision encoder turns the image into a grid of patch embeddings and a projector maps them into the language model's embedding space, where they sit in the sequence like words. So images consume context: several hundred tokens at moderate resolution, well over a thousand for a tiled high-resolution page. Where a model exposes a resolution or tile limit it is a real dial — halving resolution on a task that does not need fine detail can quarter the cost.

23. B ***3. Inside the files***

A converter that does not recognise an architecture may map it onto the closest one it knows. The file loads, it produces text, and it is subtly wrong — which is far harder to notice than a crash. The other two silent conversion defects are a chat template replaced with a default, and an end-of-turn token lost so generation never stops. Prefer official conversions, and verify a converted file against the original on the same five prompts at temperature zero.

24. C ***3. Inside the files***

The cap is the backstop that works regardless of which stop condition failed. It matters most with reasoning models, where a loop inside the thinking block can emit tens of thousands of tokens on a question that never gets answered. Set it to about twice the longest legitimate answer you expect, and treat routinely hitting it as data: either the task needs more room or something is looping.

25. D ***4. Making it run on the machine you have***

Fit the constraints first and measure what survives. The memory budget on the machine that will really run it, the p95 latency a user will actually feel, requests per day, whether the licence permits what you are shipping, and whether the data may leave the building. Most shortlists collapse immediately, which is the point — you then spend your measurement effort on two or three candidates rather than on thirty.

26. A ***4. Making it run on the machine you have***

At Q4 a fourteen-billion model is around nine gigabytes of weights before any cache or runtime overhead, on a machine that also has to run an operating system and a browser. That is not slow, it is not running. The honest ladder on that machine is a three to four billion model at Q4, comfortable with an eight-thousand-token context, and a seven or eight billion model at Q4 that fits but leaves nothing for anything else.

27. B ***4. Making it run on the machine you have***

Damage is not spread evenly. Long-context retrieval goes first, because finding one fact deep in a document depends on fine distinctions between key vectors. Lower-resource languages go next, because they sit in less well-populated regions of the representation with less redundancy behind them. Then multi-step arithmetic, then instruction adherence over long outputs. Short English chat breaks last, and it is unfortunately the first thing everybody tests.

28. C ***4. Making it run on the machine you have***

Layer offloading is the single most important dial on a machine with a graphics card, and speed scales roughly with the fraction of layers offloaded — half the layers on the card is much better than none. The neighbouring mistake is a build compiled without the right backend, which works and is several times slower. Measure prefill and decode separately before and after any change, rather than believing a setting helped.

29. D ***4. Making it run on the machine you have***

A tag resolves to a specific quantisation, usually the community default, and temperature, top-p and repetition penalty come from each model's own configuration rather than from you. Two models on different defaults are not a controlled comparison. Record the context length, the quantisation and the sampler alongside any result you intend to act on; most irreproducible local benchmarks are irreproducible because one of those three was a default nobody wrote down.

30. A ***4. Making it run on the machine you have***

Decoding is bounded by memory bandwidth: producing one token means reading every active weight once. In a batch those weights are read once for all the sequences in it, so eight users cost barely more bandwidth than one. That is the entire economic basis of hosted inference, and it is also why continuous batching and paged attention pay off — they keep the batch full and stop the cache being over-reserved.

31. B ***4. Making it run on the machine you have***

Free sessions end, and the constraint that hurts is not memory. Three habits make a long job survivable: append one line per completed item immediately, checkpoint to storage that outlives the session rather than to its local disk, and make the job resumable from the last checkpoint. With them a disconnection is a delay; without them it is a loss. These are the same disciplines that matter on paid infrastructure.

32. C ***4. Making it run on the machine you have***

The constraint is permission rather than hardware: a process that exceeds the per-application cap is terminated without ceremony. That budget means a one to three billion model at four bits, which makes a phone a private text-transformation engine — rewriting, tagging, extraction, short offline translation — rather than an assistant that knows things. Thermal throttling is the second surprise, halving sustained speed after a minute or two.

33. D ***5. Getting good output***

Greedy decoding takes the highest-scoring token every time, so one bad deterministic step inside a long chain of thought repeats forever. A little sampling breaks the loop. This surprises people who have correctly learned that temperature zero is the safe default for extraction and classification — the rule is real and this is the documented exception, which is why the model card is the authority rather than habit.

34. A ***5. Getting good output***

A field produced early conditions everything generated after it. Evidence before the answer lets the model work before it commits; evidence after the answer is a justification of a decision already made. It is a real and easily measured effect, and it sits alongside two other schema rules: always provide a way to express absence, and keep the object flat, because deeply nested designs are much harder for small models.

35. B ***5. Getting good output***

Different instruction-tuning data taught different conventions: how strongly a system prompt binds, whether headings are natural, how a numbered list of rules is treated, whether the model prefers to explain before answering. Prompts therefore do not transfer. Version one per model next to the model revision, and when you compare two models, tune a prompt for each first — otherwise you are comparing a tuned system against an untuned one.

36. C ***5. Getting good output***

Tool calling is a format the model was trained to emit plus a parser on the server written to recognise it. Families differ: a tagged block for some, a JSON object with particular keys for others, a call-like syntax for others still. Nothing is broken when they do not line up; the parser simply does not match. Naming the right parser fixes it, and asking for constrained JSON you dispatch yourself removes the variable entirely.

37. D ***5. Getting good output***

Selection accuracy falls off sharply as the list grows, and beyond about ten tools a small model is close to guessing between similar ones. A routing step — a cheap classifier or a retrieval step that picks five candidates — puts the model back inside the range where it works. The same pattern recurs throughout: reduce the number of decisions the model makes rather than enlarging the model.

38. A ***5. Getting good output***

Compounding is unforgiving: reaching ninety per cent overall across ten steps needs ninety-nine per cent per step, which no small model provides on an open-ended decision. Five deterministic steps with two model calls at ninety-two per cent is eighty-five per cent overall rather than forty-three. Most of what gets written as an agent loop is a workflow with a fixed shape, and encoding that shape in code is the larger win. Verification after each step is the other.

39. B ***5. Getting good output***

To the tokenizer, and often to the model, the two scripts are different problems, and which one a model handles depends on what its corpus contained — web forums are heavily romanised, formal publishing is not. Code-switching makes it sharper still: real Hinglish mixes English words into Hindi grammar in Roman script, and no public benchmark covers it. If your users write that way, your evaluation set must too.

40. C ***5. Getting good output***

Nothing inspects an open model's output and blocks it. When a model declines, it is generating a refusal, because preference tuning made that the likeliest continuation. Interpretability work has found the behaviour is mediated substantially by one direction in activation space, which can be identified and subtracted without retraining. Safety tuning is therefore a default, not a control, and a boundary your product needs belongs outside the model.

41. D ***6. Adapting a model to your work***

Two hundred million input tokens a month, paid to re-send the same instructions, is often the strongest financial case for a tune. The other genuine signals are having three hundred or more real examples, needing a smaller model than the task seems to need, and a format that must hold over long outputs. Note what is absent: fine-tuning changes form reliably and knowledge badly, and it adds no reasoning capacity.

42. A ***6. Adapting a model to your work***

Loss on the user's tokens teaches the model to produce user messages, which is a real and quiet defect: the run completes, the loss curve looks ordinary, and the model starts writing both sides of the conversation. Standard trainers mask the prompt correctly when configured to, and this is one of the things worth checking explicitly rather than assuming, along with which chat template the trainer applied.

43. B ***6. Adapting a model to your work***

The original paper adapted query and value projections, but most of a transformer's parameters are in the feed-forward stack, so leaving it untouched spends the budget in the wrong place. Targeting all the linear layers measurably outperforms for the same number of trainable parameters. Raising rank instead is the common wrong move: higher rank has more capacity to overfit a small dataset and forgets more.

44. C ***6. Adapting a model to your work***

Supervised LoRA tuning runs at around one to two parts in ten thousand; preference tuning wants roughly a tenth of that. Preference methods adjust a model that already does the task, so the intervention is meant to be small. The other damage to watch for is verbosity drift — human raters and judge models both prefer longer answers — capability loss from too many epochs, and reward hacking on surface format.

45. D ***6. Adapting a model to your work***

DPO needs matched triples — one prompt, a chosen answer and a rejected one — which means somebody deliberately comparing two answers to the same input. A thumbs button does not produce that. KTO learns from binary judgements, so the feedback real products already collect is usable as it stands. That practical difference matters more than the small quality gap between the methods.

46. A ***6. Adapting a model to your work***

Agreement with the teacher measures how well the student copies, including copying the teacher's mistakes. Only a labelled held-out set measures whether either is any good. The student also inherits the teacher's style — the headings, the hedging, the closing summary — which makes it read as more capable than it is, so anybody judging by skimming five outputs will overrate it.

47. B ***6. Adapting a model to your work***

Mixing general instruction examples into the training set anchors the model while it learns your task, and the usual finding is almost all of the task gain with a small fraction of the regression. Fewer epochs and a lower rank help for the same reason: less total movement. Raising rank does the opposite, and merging fixes nothing — it makes the change permanent rather than reversing it.

48. C ***6. Adapting a model to your work***

A fine-tuned deployment is five things pinned together: base revision, adapter, prompt, sampler and two evaluation numbers. Change any one and the behaviour changes, while the release notes and the dashboards still describe the old combination. Keep the block in the repository beside the code, and gate any promotion on the task metric and the regression score measured at identical settings.

49. D ***7. Judging models and claims***

An arena measures what people pick when shown two answers, which is preference rather than correctness, and preference rewards style. That is why style-controlled variants of these rankings are the ones worth reading. Two further cautions: models are often tested under anonymous codenames and only the results a lab likes get claimed, and adjacent entries routinely have overlapping confidence intervals, which is a tie displayed in an order.

50. A ***7. Judging models and claims***

Ambiguous questions, wrong answer keys and unanswerable prompts put a noise floor under every score on that benchmark. When two models are within two or three points, a substantial share of the difference is that floor rather than either model. The practical rule from the same reasoning appears again on your own set: with forty examples, act on gaps above about ten points and treat five as nothing.

51. B ***7. Judging models and claims***

Shot count and chat template are the two biggest causes and both move a score by several points. Answer extraction and the harness version come next, since task definitions get fixed between releases and a score from last year's version is not comparable with this year's. Batch-size non-determinism is real and tiny. Pin the harness commit beside the model revision, report both, and always log the samples.

52. C ***7. Judging models and claims***

A comparison is honest when everything except the model is held constant, and that is hard to guarantee across two codebases with two logging conventions. One task file means your number and the public ones come out of the same run, at the same shot count, with the same decoding settings, logged the same way. It takes about an hour and turns an assembled comparison into a reproducible one.

53. D ***7. Judging models and claims***

Two variables changed together cannot be attributed, and these two are commonly changed together because both save memory. Four configurations — full weights with full cache, quantised weights with full cache, full weights with quantised cache, and both — settle it. Long-context items are where cache quantisation shows up, because subtle key distinctions are exactly what lets a model find a fact far back.

54. A ***7. Judging models and claims***

With forty examples the interval around eighty per cent accuracy is roughly plus or minus twelve points, and the interval on the difference between two models is wider still. A five-point gap is nothing. Write that sentence into the report yourself, because whoever reads it will otherwise treat the two numbers as a decision — and the honest alternatives are to collect more examples or to decide on something else.

55. B ***7. Judging models and claims***

Shadowing costs nothing but compute: live inputs go to both models, only the old model's responses are served, and both outputs are stored. After a day you have thousands of paired comparisons on exactly the traffic you care about, and reading two hundred of them tells you more than any held-out set. It is the highest-value step in a model swap and the one most often skipped in favour of going straight to a canary.

56. C ***7. Judging models and claims***

Tuning and measuring on the same items reports how well you fitted them, which is always flattering and never transfers. Split the set: about fifteen development examples to iterate against, and twenty-five held out and touched exactly once at the end. The same discipline appears in the fine-tuning block as deduplicated splits, and in prompting as holding back items you have not looked at.