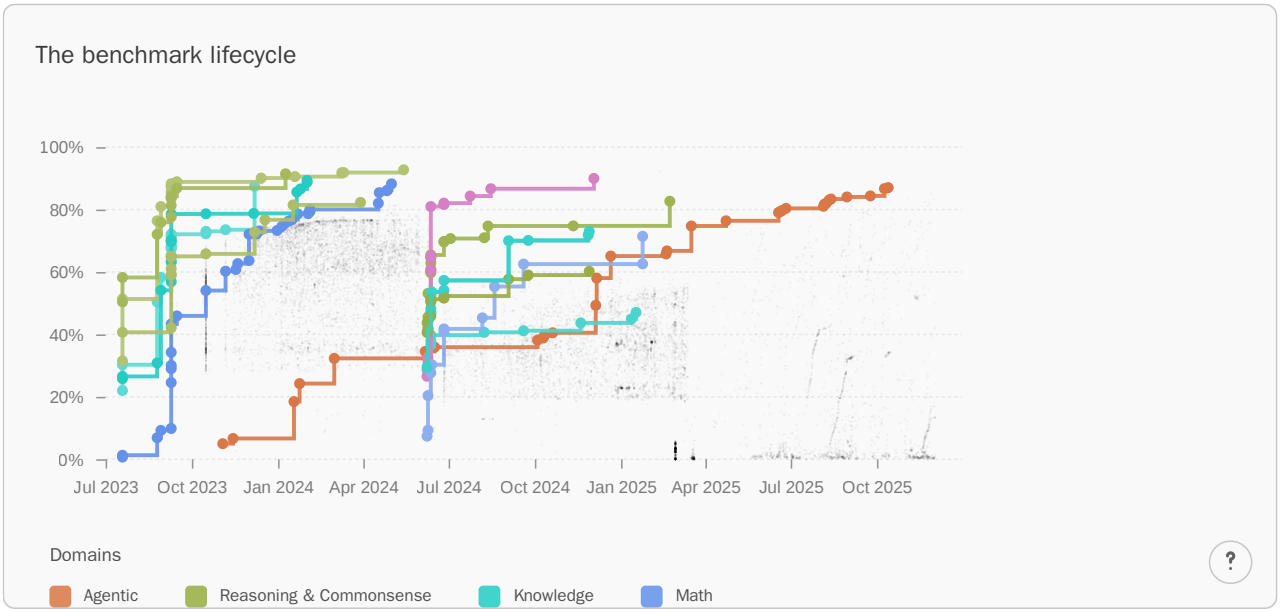
The LLM Evaluation Guidebook



All the things you could want to know about LLM evaluation based on our experience scoring 15000 models over 3 years

AUTHORS

Clémentine Fourrier, Thibaud Frere, Guilherme Penedo, Thomas Wolf

AFFILIATION

Hugging Face

PUBLISHED

Dec. 03, 2025

What is model evaluation about?

As you navigate the world of LLMs — whether you're training or fine-tuning your own models, selecting one for your application, or trying to understand the state of the field — there is one question you have likely stumbled upon:

How can one know if a model is *good*?

The answer is (surprisingly given the blog topic) evaluation! It's everywhere: leaderboards ranking models, benchmarks claiming to measure *reasoning*, *knowledge*, *coding abilities* or *math performance*, papers announcing new state-of-the-art results…

But what is evaluation, really? And what can it really tell you?

This guide is here to help you understand it all: what evaluation can and cannot do, when to trust different approaches (what their limitations and biases are too!), how to select benchmarks when evaluating a model (and which ones are relevant in 2025), and how to design your own evaluation, if you so want.

Through the guide, we'll also highlight common pitfalls, tips and tricks from the Open Evals team, and hopefully help you learn how to think critically about the claims made from evaluation results.

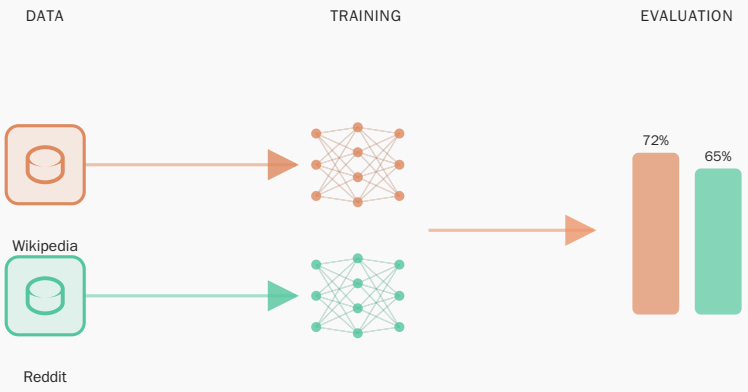
Before we dive into the details, let's quickly look at why people do evaluation, as who you are and what you are working on will determine which evaluations you need to use.

The model builder perspective: Am I building a strong model?

If you are a researcher or engineer creating a new model, your goal is likely to build a strong model that performs well on a set of tasks. For a base model (training from scratch), you want the model to do well on a general tasks, measuring a variety of different capabilities. If you are post-training a base model for a specific use case, you probably care more about the performance on that specific task. The way you measure performance, in either case, is through evaluations.

As you experiment with different architectures, data mixtures, and training recipes, you want to make sure that your changes (choosing different training data, architecture, parameters, etc) have not "broken" the expected performance for a model of these properties, and possibly even improved it. The way you test for the impact of different design choices is through ablations: an ablation is an experiment where you typically train a model under a specific setup, evaluate it on your chosen set of tasks, and compare the results to a baseline model. Therefore, the choice of evaluation tasks is critical for ablations, as they determine what you will be optimizing for as you create your model.

Ablation example



Say you want to compare dataset A and dataset B (for example, Wikipedia vs Reddit) to see how they affect model performance. You train models under the same setups on each, then evaluate and compare the scores on benchmarks.

For base models, one would typically resort to selecting standard benchmark tasks used by other model builders (think the classic list of benchmarks that are always reported when a new model is released - we'll have a look at those below). For a specific use case, you can either use existing evaluation tasks if they are available — and you likely will want to take a good look if they are not "standard" — or design your own (discussed below). As you will likely run a lot of ablations, you want the evaluation tasks to provide strong enough signal (and not just meaningless noisy results) and you want them to run cheaply and quickly, so that you can iterate fast. Through ablations, we are

also able to predict the performance of bigger models based on the perfomance on smaller ones, using scaling laws.

Besides ablations for experiments, you will likely also want to run evaluations on intermediate checkpoints as your model is training, to ensure it is properly learning and improving at the different tasks, and does not start regressing due to spikes or other issues. Finally, you want to evaluate the final checkpoint so that you can announce that your model is SOTA when you release it.

> Small caveat
>
> Despite often grandiose claims, for any complex capability, we cannot at the moment just say “this model is the best at this”, but should instead say “this model is the best on these samples for this specific task that we hope are a good proxy for this capability, without any guarantee”.

(You can still claim you are SOTA, just keep the caveat in mind.)

The model user perspective: Which model is the best on <task>?

You want to use a model someone else trained for your specific use case, without performing additional training, or maybe you will perform additional training and are looking for the best existing model to use as a base.

For common topics like math, code, or knowledge, there are likely several leaderboards comparing and ranking models using different datasets, and you usually just have to test the top contenders to find the best model for you (if they are not working for you, it’s unlikely the next best models will work).

You could want to run the evaluation and comparisons yourself (by reusing existing benchmarks) to get more details to analyse on the model successes and failures, which we will cover below.

Similarly to model builders hillclimbing a specific capability, for less common topics, you might need to think about designing your own evaluations, which is detailed in our last section.

> 🎯 Takeaways
>
> - Model builder: You need fast, high-signal benchmarks that cover the domains/capabilities you care about and can be run repeatedly during ablations.
> - Model user: You need benchmarks that match your specific use case, even if that means creating custom ones.

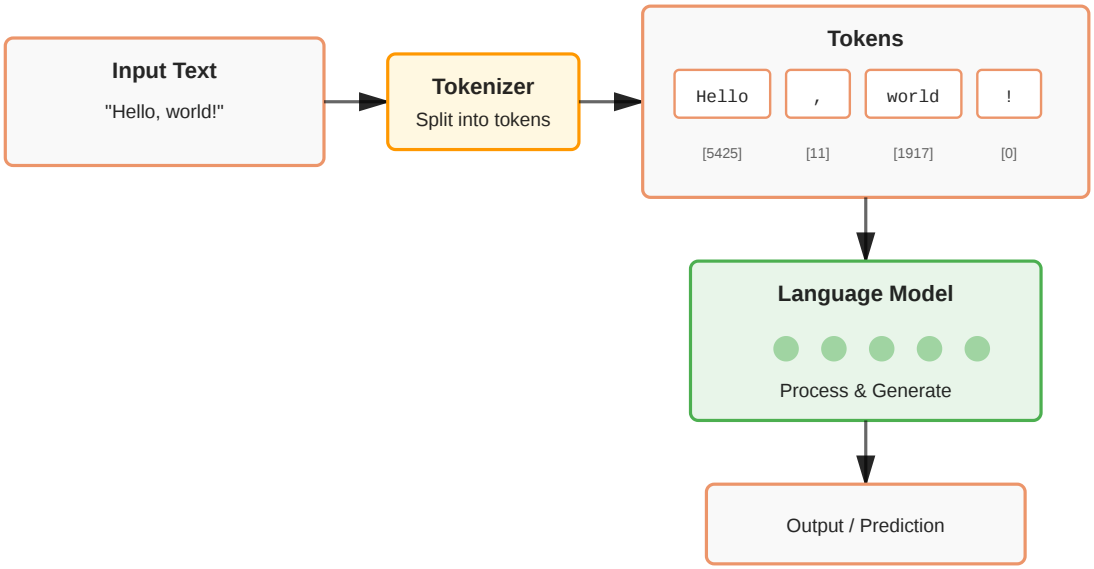
> What about measuring AGI?
>
> We are strongly missing any kind of good definitions and framework on what intelligence is for machine learning models, and how to evaluate it (though some people have tried, for example Chollet in 2019 and Hendrycks et al this year). Difficulty in defining intelligence is not a problem specific to machine learning! In human and animal studies, it is also quite hard to define, and metrics which try to provide precise scores (IQ and EQ for example) are hotly debated and controversial, with reason.
>
> There are, however, some issues with focusing on intelligence as a target. 1) Intelligence tends to end up being a moving target, as any time we reach a capability which was thought to be human specific, we redefine the term. 2)

> Our current frameworks are made with the human (or animal) in mind, and will most likely not transfer well to models, as the underlying behaviors and assumptions are not the same. 3) It is kind of a useless target too - we should target making models good at specific, well defined, purposeful and useful tasks (think accounting, reporting, etc) instead of aiming for AGI for the sake of it.

LLM basics to understand evaluation

Now that you have an idea of why evaluation is important to different people, let's look at how we prompt models to get some answers out in order to evaluate them. You can skim this section if you have already done evaluation and mostly look for the notes and sidenotes.

In this section, we'll look at two steps for models: how input is preprocessed to be given to the model (`tokenization`), and how the model generates a prediction from it (`inference`).



Tokenization

The input text (called a *prompt* at inference) is first split into *tokens*, small units of texts (which can be one or several characters, up to the word level) each associated with a number. The whole range of tokens a model can parse is called its *vocabulary*.

BASICS OF TOKENIZATION: WHY AND HOW DO WE TOKENIZE TEXT?

Since large language models are actually big mathematical functions, they eat numbers, not text.

Say you want to transform a sentence to numbers. You first need to decide how to cut your sentence into small pieces, then map every small piece to a number; this is *tokenization*.

In the past, people would try to map each character of a text with its index in a alphabet (`a` -> 1, `b` -> 2, etc) which is called *character based tokenization* (you split between characters). On the other end of the spectrum, people also tried to map each word with its index in a dictionary (`a` -> 1, `aardvark` -> 2, `ab` -> 3, etc) which is called *word based tokenization* (you split on spaces, if your language has spaces - if not, it's a bit harder).

Both these methods share a strong limitation: they remove information from the input text. They erase semantic connections that you can see from word shape (ex: `dis similar`, `similar`, `similar ity`, `similar ly`), information we would like our model to retain, so it connects related words together. (Plus, what happens if you suddenly have a completely new word in input? It gets no number, and your model can't process it 😔)

Some people therefore had the idea to cut words into sub-words, and assign index to these sub-words (`dis`, `similar`, `ity`, `ly`)!

This was initially done using morpho-syntactic rules (*morpho-syntax* is like the grammar of word creation). Now most people use byte pair encoding (BPE), a smart statistical method to create the sub-words automatically depending on their frequency in a reference text.

So as a summary: tokenization is a way to map small units of texts (which can be one or several characters, up to the word level) to numbers (similar to an index). When you want to process text, your input text (called a *prompt* at inference) is split into these *tokens* by a tokenizer. The whole range of tokens a model or tokenizer can parse is called its *vocabulary*.

> 📚 Going further: Understanding tokenization
>
> - ⭐ Explanation of different tokenization methods in the 🤗 NLP Course
> - ⭐ Conceptual guide about tokenization in the 🤗 doc
> - Course by Jurafsky on tokenization (and other things) - skip to 2.5 and 2.6

> 📚 Going further: Byte Pair Encoding
>
> I would strongly recommend reading a longer explanation on how BPE works, as it's really a base of modern LLMs.
>
> - ⭐ Explanation of BPE in the 🤗 NLP Course
> - BPE Paper (for text, as the method existed before in other fields)

Building a tokenizer requires making more choices than one would expect. For example, to tokenize numbers, you don't want to use a basic BPE, but do you only index 0 to 9, and assume all other numbers will be compositions of digits? Do you want to store numbers up to, say, one billion, individually?

Current well known models display a range of approaches to this, but it's unclear what works better to allow mathematical reasoning. This will affect some mathematical evaluation (and is the reason why almost no evaluation is pure arithmetics).

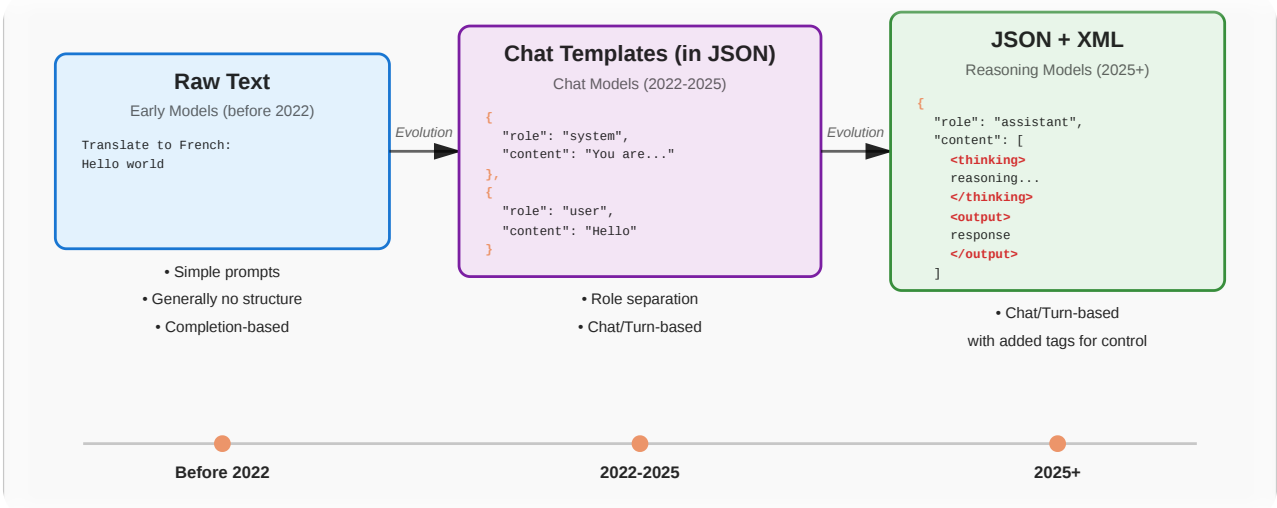
> 📚 Going further: Tokenizing numbers
>
> - ⭐ A nice visual demo by Yennie Jun of how tokenizers of Anthropic, Meta, OpenAI, and Mistral models split numbers

- Small history by Beren Millidge of the evolution of number tokenization through the years

HOW TOKENIZATION CAN MESS UP YOUR EVALUATION

Managing fine-tuned models, system prompts and chat templates

Pre-2022, models used to simply be pretrained: text in, text out, nothing else. Then, we got instruction tuning and chat models in 2023, and in 2025 reasoning models. This means that we went from using raw text to using more and more formatting.



This means a number of models are going to perform terribly if you do not make sure to:

1. respect the format the model expectes
2. adds a system prompt at the very beginning of inference if your model requires one
3. remove the thinking trace from reasoning models answers before processing them (you can usually regex to remove what's between the `<think>` tags)

⚡ Critical: Chat templates and tokenization

Different tokenizers behave differently with spacing and special tokens. See this visualization showing how spacing, tokenization, and templates interact. Never assume tokenizers behave identically!

Paying attention to start and end of sentence tokens

Some pretrained models, like the `Gemma` ones, are extremely sensitive to the inclusion of start of sentence tokens at inference. You might need to do a couple of experiments to see if that happens for you, and add these tokens manually when evaluating if they are not in your dataset.

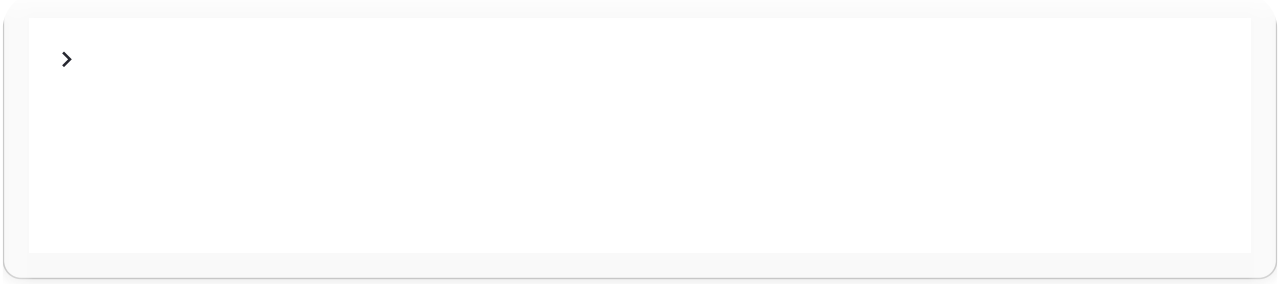
You can also encounter some issues where your model won't stop on an end of sentence token like you would expect. Code models usually have been trained with `\n\t` as a single token. This means that when generating text, they will often generate `\n\t` in one step. A task which defines `\n` as an end of sentence token (= to stop the generation) will let the model continue generating after a `\n\t`, if predicted as one token, since it's not the same as `\n`. But you would actually still want the model to stop. In these cases, you either need to update your end of sentence tokens, or define a mechanism to backtrack on the character representation of the latest tokens to stop (and cut) the generation a posteriori.

Multilinguality and tokenization

When looking at multilingual evaluations, you'll encounter two issues.

First, as some languages do not always use spacing as a word separator (Korean, Thai, Japanese, Chinese, to cite a few), they will require language specific tokenizers to be split properly, else it will affect their scores on metrics such as BLEU, F1 scores, etc.

Then, tokenizers in general might be unfair to non-English languages. When training a BPE tokenizer, you use data from the different languages you want to cover, but most of the time, though, this data is unbalanced between languages (with, for example, an order of magnitude more English than Thai, or Burmese). Since BPE tokenizers create their vocabulary tokens based on the most frequent words seen, most of the long tokens will be English words - and most of the words from the less frequent languages will only be split at the character level. This effect leads to an unfairness in multilingual tokenization: some (less frequent, or *lower-resourced*) languages require orders of magnitude more tokens to generate a sentence of equivalent length as English.



Very nice demo by Yennie Jun on tokenization issues across languages

If you are in this case, the number of tokens that the model is allowed to generate for an evaluation should also be language dependent, as not all languages are tokenized in similar amount of tokens.

> 📚 Going further: Language and tokenization
>
> - ⭐ A beautiful breakdown and demo by Yennie Jun on tokenization issues across languages: The breakdown in itself is very clear, and the embedded space comes from her work.
> - ⭐ A demo by Aleksandar Petrov on unfairness of tokenization: I recommend looking at `Compare tokenization of sentences` to get a feel for the differences in cost of inference depending on languages

Inference

Now that we know how to convert our input text into something the LLMs can parse, let's look at how models process this text.

From this input text, the LLM generates a probability distribution of the most likely next tokens over all the vocabulary. To get a continued generation, we can take the most probable token (give or take some added randomness to get more interesting outputs) as the next one, then repeat the operation, using the new token as the end of the prompt, etc.

> 🎯 Two main evaluation approaches
>
> Log-likelihood evaluations: Given a prompt and one (or several) answers, what is probability of said answer(s) for my model?
>
> Generative evaluations: Given a prompt, what text does my model generate?
>
> Choice depends on your task (as we'll see below) and on your model: most models under APIs do not return the logprobabilities, so you'll need to use generative evaluations systematically to evaluate them.

LOG-LIKELIHOOD EVALUATIONS

For log-likelihood evaluations, we want the conditional probability of one or several choices given a prompt - in other terms, what is the likelihood to get a specific continuation given an input? So:

- we concatenate each choice with the prompt, and pass them to our LLM, which outputs the logits of each token depending on the previous ones
- we only keep the last logits (associated with the choice tokens), and apply a log softmax to get log-probabilities (where the range is `[-inf, 0]` instead of `[0-1]`)
- we then sum all individual tokens log probabilities to get the overall choice log probability
- we can finally apply a normalization based on choice length

This allows us to apply one of the following metrics:

- get the preferred answer of a model among several choice, like in the above picture. (*However, this can advantage scores of models which would have, freely, generated something else, like `Zygote` in the picture.*)
- test if a single choice has a probability above 0.5
- study model calibration. A well calibrated model is a model for which the correct answers have the highest probabilities.

> A multiple choice question answer can be expressed as a free form generative evaluation too! For this reason, you'll sometimes see a mention of the task formulation.
>
> There are three common task formulations:
>
> - Multiple choice format (MCF): we compare the likelihood of choices indices, where choices are explicitly presented in the prompt and prefixed with A/B/C/D (as in MMLU)
> - Cloze formulation (CF): we compare the likelihood of different choices without providing them in the prompt
> - Freeform generation (FG): we evaluate the accuracy of greedy generation for a given prompt
>
> FG requires substantial latent knowledge and is usually too difficult for models during short pre-training ablations. For this reason, we typically focus on multiple choice formulations (MCF or CF) when running small-scale ablations. However, for post-trained models, FG becomes the primary formulation since we're evaluating whether the model can actually generate useful responses. However, research has also shown that models struggle with MCF early in training, only learning this skill after extensive training, making CF better for early signal. We thus recommend using CF for small ablations, and integrate MCF in the main run as it gives better mid-training signal once a model has passed a threshold to get sufficiently high signal-over-noise ratio for MCF. A quick note also that, to score a model's

answer in sequence likelihood evaluations like CF, we compute accuracy as the percentage of questions where the the correct answer has the highest log probability normalised by character/token count. This normalisation prevents a bias toward shorter answers.

Should you tokenize the context with the choices always?

GENERATIVE EVALUATIONS

For a generative evaluation, we want the text generated by the model given an input prompt.

It is obtained in an auto-regressive way: we pass the prompt to the model, look at the most likely next token, select it as being the model's "choice first token", then repeat until we reach an end of generation condition (maximum length, special token to stop the generation, etc). All the tokens generated by the model are consider its answer to the prompt.

We can then compare this generation with references and score the distance between both (using either simple metrics like exact match, more complex metrics like BLEU, or models as judges).

📚 Going further

- ⭐ Blog on several ways to evaluate MMLU , by my team at Hugging Face. I recommend reading it if you want to delve deeper into the differences between multi choice log-likelihood evaluations and generative ones, including what it can mean with respect to score changes (The above illustrations come from the blog and have been made by Thom Wolf)
- ⭐ A beautiful mathematical formalization of the above inference methods, from EleutherAI. Go to the Appendix directly.

Evaluating with existing benchmarks

Now that you've gotten (re)acquainted with required basics on how tokenization and inference work, and what are the caveats when doing evalution, let's look at actual benchmarking! We'll first do a small tour of 2025 evaluations, then discuss what to look at in a benchamrk, and why you probably can't reproduce announcements scores. Lastly, we'll cover the special case of selecting good benchmark to evaluate training with the FineWeb team.

> ⚠️ Important concepts
>
> In this section, you'll see two concepts mentionned quite a lot: contamination and saturation.
>
> Saturation is when model performance on a benchmark passes human performance. More generally, the term is used for datasets that are no longer considered useful, as they have lost discriminative power between models.
>
> *If all models have close to the highest possible score on your evaluation, it's no longer a discriminative benchmark. It's similar to evaluating high school students on pre-school problems: success tells you nothing (though failure is indicative).*
>
> Contamination is when an evaluation dataset ended up in the training dataset of models, in which case the performance of models is artificially inflated, and does not reflect real world performance on the task.
>
> *It's a bit like evaluating a student on questions it already knows in advance.*

Benchmarks to know in 2025

You can evaluate specific capabilities on their own - it's usually quite interesting to get signal when training, or when comparing base/pretrained models. (However, if you select and validate your training methods with the following evaluations, reporting on them on the final model is slightly biased as you have already oriented your training method towards good results on them).

> Feel free to skim this section if you're not very familiar with evaluation yet, and come back to it once you need to find a dataset for a specific capability :)

REASONING AND COMMONSENSE

Reasoning and commonsense datasets are often "historic" datasets, built in the age of BERT and embeddings model, before the LLM craze. They were quite challenging at the time (especially because they were often adversarially built for models of the time), but now they are 1) too easy 2) contaminated/saturated, and should only be used for ablations or as pretraining evaluations. The bigger datasets also sometimes contain errors or low quality questions as they tend to have been built through Amazon Mechanical Turk in order to scale up fast and at low cost (what is now done by using LLMs to generate evaluation questions).

ARC (2018) (not to confuse with ARC-AGI) is a grade school science MCQA dataset built from human tests. The choices were selected adversarially for word co-occurence systems at the time. It has several subsets, the higher quality `challenge` one is still in use today for pretraining. WinoGrande (2019) is a crowdsourced (mechanical turk

+ validation) pronoun resolution/fill in the blank dataset, using adversarial pairs of items to trick models. Both these datasets have been quite hard for models until 2022 to 2023.

A number of historic datasets are looking specifically at reasoning requiring some sort of commonsense understanding and grounding. HellaSwag (2019) requires LLMs to select the correct next sentence in a list of adversarial choices, where the text comes from captions in ActivityNet and from tutorials in Wikihow. (It's the follow up of a dataset called Swag). As most sentences come from tutorials or descriptions of activities, they often require physical commonsense grounding to solve. In the same vein, CommonsenseQA (2018) is a dataset of commonsense MCQA built from ConceptNet - annotators write questions, then use conceptually close distractors as options. PIQA (2019) is specifically looking at physical commonsense questions (created from examples from Instructables.com, with again adversarial choices from semantic perturbations or rewriting). OpenBookQA (2018) provides open book facts to help answer MCQA questions - however, these questions also require latent common sense knowledge.

A more recent cool reasoning dataset is Zebra Logic, using logic puzzles to test model reasoning capabilities. Their methods allows for infinite generation of puzzles, so little contamination.

KNOWLEDGE

The main evaluation dataset for knowledge has been MMLU (2020). It reached saturation/contamination, and after more in depth examination, a number of issues were identified: incomplete questions referring absent documents, incorrect ground truths, ambiguous questions, and blatant americano-centrism in the topics chosen. It was therefore cleaned in MMLU-Redux (2024), extended with more complex questions and more answers in MMLU-Pro (2024, the main replacement used by the community at the moment), and translated/annotated for cultural bias in Global-MMLU (2024). These are used mostly for pretraining evaluations and ablations.

For post training, people look at harder high quality knowledge dataset. GPQA (2023), custom PhD level questions in biology/chemistry/physics, made to be answerable by PhD students in the correct domain and not otherwise. The most used subset is the `diamond` one, but since its publication in 2023 it has also started reaching contamination.

Last but not least, the pompously named but very high quality Humanity's Last Exam (2024) contains 2.5K crowdsourced questions by experts in their field, across domains. It is mostly private, and questions require both complex knowledge and reasoning. It has not been broken yet, and it's imo a cool dataset. The only issue is that since there is no way to get a model scored fast, people now evaluate against it by using an LLM judge to assess their answers, insted of checking against ground truth, so it's one of these evaluations where you'll get really uncomparable results in the wild.

However, though testing models for the raw quality of their latent knowledge made a lot of sense a couple years back (and is still interesting while training to test model quality, with evals like MMLU-Pro during pretraining and GPQA/HLE for post training), I think we will slowly phase out of benchmarks such as this in the next years, for 2 reasons.

1. They are becoming more and more indecipherable for humans: questions are becoming so complex that it's almost impossible for non experts to understand what performance on each question means (and to make sure the datasets themselves do not contain mistakes)
2. Now that our models are connected to tools, such as internet access, latent knowledge evaluations are increasingly becoming web search and retrieval evaluations, so they make less sense as such. In short, we're moving from closed book to open book evaluations. As a comparison, in the French school system, you get

closed books examinations in high school, but as you enter university, it's often assumed that you will get access to databases, internet, and scoring becomes less about what you learnt by heart, and more about how you reason given free access to information. I believe this is also a change we will see in LLM evaluation with the increase of model capabilities.

MATH

Math evaluation datasets have been used as proxies for reasoning and logic benchmarking, independently of, obviously, also checking if models can solve math problems.

The two reference math evaluation datasets were GSM8K (2021), containing grade school math problems and MATH (2021), an aggregation of Olympiad problems present on the web, which reached saturation/contamination in the last years. The former was extended by GSM1K (2024), a recreation with 1K new problems, to test which models were contaminated on the former, GSM-Plus, a rewriting of models with adversarial changes (distractors, numerical variations, and so forth) and GSM-Symbolic (2024), less used, but a very interesting re-writing of GSM8K as problem templates, to prevent contamination: problems can be regenerated ad infinitum.

Community has now been focusing on using:

- The follow ups to MATH, either MATH-500 (a representative subset of 500 problems sampled to avoid overfitting) and MATH-Hard (only the 500 hardest questions)
- AIME (24, 25), american olympiad datasets for high schoolers, taken as is at publication. These datasets are interesting because, since they are made of problems renewed every year with equivalent difficulty, they allow testing for contamination by comparing results at publication with results on the previous year's dataset
- Math-Arena, an up to date compilation of competitions and olympiads actualised regularly (it contains AIME25, but a lot of other competitions too!)

Most of these datasets are actually no longer "that hard", since they stop at grade school level (even though GSM-Symbolic allows to generate problems with more recursion levels, making them synthetically harder). On the other side of the spectrum, FrontierMath (2024) was an attempt at providing considerably harder math problems, written individually by mathematicians for the occasion. The dataset was theoretically private (but it appeared OpenAI has had access to parts of the dataset - such a shame). Humanity's Last Exam (2025) (introduced in the knowledge section) also contains interesting "made for the occasion" math problems requiring complex reasoning (notably some theorem proving).

I would personally use AIME25 and MATH-500 for pretraining evaluations, and the Math-Arena for post training.

CODE

Since agents need to interact with tools, they need coding abilities, either to call tools directly if they are code agents, or understand how to debug tool output in case of problems (for code and json agents both, see the difference here). Coding evaluation sets are also good proxies for reasoning.

Historically in 2021, code evaluation sets were MBPP, 1K crowdsourced Python only entry-level programming problems, APPS, 10K code generation problems curated from programming interviews and sharing websites, and HumanEval, introduced with the Codex model, which contrary to the previous is made of "specifically made for the release" problems, which was super neat then! It also came with a sandbox to avoid problematic code execution on

the evaluator's machine. (Last thing this paper introduced was an estimator for `pass@k`, which before that was computed with a literal check on whether an evaluation was a success more than k times on n).

The EvalPlus (2023) team made HumanEval+ and MBPP+, extensions of the former, by adding more test cases and fixing bugs in the original datasets as well as adding more inputs. EvoEval (2024) also introduced a variation on HumanEval by semantically rewriting the problems and adding difficulty labeling.

For final models, you might want harder or uncontaminated problems.

LiveCodeBench (2024) follows a similar "grabbing from leetcode websites" approach, but is very interesting because it stores the problem date, to compare model performance on problems created before and after they finished training. This was an excellent contamination free benchmark, and I'm looking forward to an update!

AiderBench (online since end of 2024 I think?) also uses data from existing coding websites (Exercism to be specific), but goes beyond problem solving by testing specifically code editing and refactoring.

For post training, you want more holistic evaluations, and a couple benchmarks moved beyond evaluation on standalone problems, which were not evaluating complex coding abilities. RepoBench (2023) tests repository level auto completion systems in Python or Java, using code from Github as source. It was built by masking random lines in code bases and asking for completions, either a cross file or in file function, and defines several tests level (retrieval, completion, a combination).

SweBench (2024) is a more well known and complete version of this, also using github, but this time testing if models can solve existing issues, so logic understanding, cross file editing and execution, long context reasoning, etc.

CodeClash (2025) is the coding version of an arena, where models write code which competes against other models code, edit, and iterate.

At this time, I would recommend following LiveCodeBench, AiderBench and the higher quality subset of SWE-Bench (SWE-Bench verified), and reading the METR report on actual code assistant usefulness.

LONG CONTEXT

To correctly interact with users over a long discussion, without losing track, you need good long context management. (Funny to think that 3 years ago, maximum context lengths for models were 2048 tokens, when now we're largely at 128K and beyond).

The evaluation which started testing this in 2023 is probably NIAH, (Needle in a Haystack), where you place a random fact in a long unrelated text and ask the model to retrieve it. It provides a neat framework to evaluate where in the context a model is most likely to forget stuff, and from which context length. In 2023 models were really bad at it, in 2025 it's close to solved.

More complex long context extensions have emerged since. RULER (2024) adds multi-hop tracing (requiring the model to follow chains of variables to get the correct value), word frequency changes, and adds a QA variation of NIAH. it's also close to solved now. Michelangelo (2024, also sometimes called MRCR for multi round co reference) is also using synthetic long context data: tasks (of varying length) test whether models can reproduce precisely unique portions of the context (as well as identify if relevant information is present) and understand sequence of modifications to a text. It was then extended in the OpenAI MRCR (2025). InfinityBench (2024) is multilingual (En and Zh), and provides 100K tokens synthetic data tasks, across a variety of objectives (QA, retrieval as in NIAH, computations over very long context, ⋯). InfinityBench still provides some signal.

HELMET (2024) combines tasks and existing benchmarks to get a big single dataset with more signal: RAG and QA datasets (Natural questions, TriviaQA, PopQA, HotpotQA, Narrative QA and InfinityBench), recall (RULER and JSONKV), generation with citation (subsets of ALCE), summarisation, reranking passages (MS MARCO), in context learning (TREC, NLU, Banking77, CLINIC150). Benchmark aggregations are exhaustive but present the risk of measuring things two times : don't go testing your model against both HELMET and InfinityBench, then aggregating the results, for example, as you would run the same evaluation twice! In 2025, it still has enough discriminative power to compare models.

My favorite long context evaluations ideas are the Novel Challenge (2024), 1K true/false claims about fictional books published in the last year (by readers of said books!) requiring having read and understood the full text to answer properly, and the Kalamang translation dataset (2024), where models need to properly translate from English to Kalamang from reading a grammar book (Kalamang is such a low resource language that it has no online presence - only 200 speakers). The Kalamang translation set could notably be expanded to other low resource languages (but it would be cool to expand to use a rule based grammar checker to test generation validity to get strict accuracy instead of relying on BLEU…).

INSTRUCTION FOLLOWING

The two main instruction following datasets are IFEval (2023) and its extension IFBench (2025). IFEval is one of the smartest evaluation ideas in the last years, in my opinion: models are asked to follow formatting instructions (about keywords, punctuation, number of words/sentences, file type formatting such as markdown or html, etc). Each of these conditions can be checked with a specific parsing test: this means that this evaluation is one of the rare free form generative evaluation where you can get a strict score without relying on a model judge.

More generally, it falls into the functional correctness/unit test evaluation type, which is my personal favorite way to evaluate models. It's also very easy to regenerate or extend to prevent contamination.

Side note, but some benchmarks also test "non instruction following" (non compliance): CoCoNot (2024) notably tests if models will or won't comply with incomplete (underspecified/unclear), unanswerable (by lack of information or AI-humanizing, often hallucinations triggering), or unsafe requests. It used manual queries writing, models to write non compliants requests, then filtered to create an eval set presented as a classification problem.

TOOL-CALLING

The emergence of tools is one of the features which started moving LLMs into the agentic realm.

TauBench (2024) evaluates a model on its ability to answer a user's query in the retail and airline domains (order/book/look for products/etc). The database mimics real domain data with synthetic samples, and the model is considered correct when 1) its actions updated the database correctly and 2) it answered the user appropriately. To make this benchmark automatic, the user is mocked up by an LLM, which makes this evaluation quite costly to run and prone to errors. Despite these limitations, it's quite used, notably because it reflects real use cases well.

ToolBench (2023) require calling APIs (OpenWeather, Cat, HomeSearch, TripBooking, GoogleSheets, WebShop, Tabletop, etc) to solve 100 test cases across dataset, requiring between one and 10 tool calls to solve. Some of these APIs are mock ups and some of them are real, which makes the dataset susceptible to accidental failure. It was therefore fixed and extended in StableToolBench (2025), which introduces a general VirtualAPIServer mocking up everything to ensure evaluation stability, however relying on an LLM judge for evaluation, introducing another layer of bias.

BFCL (2025, but the benchmark actually has a couple years) evolved considerably over the year, and in its current version contains 4 subset: single turn (simple tool calls), crowdsourced real life function calls from users, multiturn conversations (to test accuracy in long context and query answering with tool calls) and agentic (web search, memory, sql data interaction). It's using a combination of Abstract Syntax Trees, execution response and state matching (is the final state the expected one) to evaluate if calls are correct. People are focusing on the v3 to test tool calling specifically, and the v4 tests web and search tool use.

Lastly, with the creation of MCPs, some benchmarks arose to test MCP oriented tool calling - however all mostly relying on model judges, and using real world APIs, which can introduce potential failure cases/lack of reproducibility due to network issues (seems like added load for website creators is not too much of an issue as the userbase of most MCP covered is big enough).

MCPBench (2025) connects LLMs to live, real world MCP servers (Wikipedia, HF, Reddit, Steam, arxiv, ⋯) with tasks requiring multiple turns to solve (created synthetically). The evaluation combines rule based checks on tool call validity and success with an LLM judge to assess if queries were properly answered.

MCP-Universe (2025) uses 11 MCP servers across varied real world topics (IRL navigation, 3D design, web search, etc). What's cool in this one is that evaluation relies on several strict evaluators, one for format correctness, and two for answer correctness: as tasks can be static (asking things that do not change) or dynamic (github stars in a repo, weather, ⋯), in the latter case answer correctness uses a task-dependant execution based evaluation framework which grabs the latest correct answer from the relevant source automatically and compares the model output to it. This is way neater than relying on LLM judge!

LiveMCPBench (2025) provides a large locally deployable collection of MCP servers to test how good models are at discriminating between tools to accomplish tasks. Best models are already reaching 80% - so we're close to saturation. However, testing if models can select proper tools in very long lists is a good use case which will be increasingly important as the web goes mcp.

(By the way, here's a cool doc on how to write good tools.)

While testing individual capabilities provides valuable signal, real-world assistant performance comes from how these capabilities combine. A model might excel at reasoning but fail when that reasoning must be integrated with tool calling and long context management simultaneously, so we need evaluations requiring the orchestration of multiple capabilities together.

ASSISTANT TASKS

I believe that assistant tasks are going to be one of the main ways to do next level evaluations: solving them requires a combination of many capabilities (long context, reasoning, tool calling, ⋯), while the benchmarks themselves provide insight on specific domains performance in a useful real world setup. They also tend to be more understandable (by the general public) than specific capabilities benchmarks. If the benchmarks are general enough, they do not check which precise tools were used, but instead if the end result is correct, as complex tasks allow several paths to success.

Real life information retrieval

GAIA (2023) kickstarted modern agentic evaluation by requiring models to use a combination of tools, reasoning and retrieval to solve real life queries (sometimes including documents). Questions were split in 3 levels, the first one now saturated and the third one still hard for models. It's also one of these benchs were numbers you find will be

spread out against evaluation methods, because people are either reporting on the public validation set or using llm judges to evaluate against the private test set (when there is a public leaderboard here).

It was later replicated in BrowseComp (2025) which tests the same thing (can a model find the adequate answer to a specific query using tools and online information) but does not guarantee uniqueness of result, as questions were constructed by starting from the result and building a question from it, with varying levels of difficulty: for example, from a specific paper to retrieve, a question will be created by combining information about metadata, for example "which paper about Topic was published at Conference with one Nationality author and two people from Entity?" However, the benchmark is probably also harder at the moment.

GDPval (2025) evaluates models on 44 occupations from the "top industries contributing to US GDP", comparing model performance with human performance using model judges.

Lastly, GAIA2 went beyond simple information retrieval, using a mock up mobile environment to test how assistants are able to answer correctly answer queries relying on chains of events and tool calls. As of now, time sensitive and deliberately noisy subsets (mocking up failing API calls) are the hardest for models, when search and execution seem extremely easy for SOTA models.

Science assistants

SciCode (2024) tests if models can solve real life scientific problems by writing appropriate scientific code, across stem fields (from biology to math/chem/⋯). Problems are drawn from real life workflows, and each core issue is decomposed in easier subproblems. For the first version, evaluation was done by scientists and a model judge - models were quite bad at it at publication (less than 5% scores) but I'm unsure where up to date results can be found.

PaperBench (2025) similarly tests if models can replicate ML research, but this time with a harder setup: given ICML high quality papers, models must reconstruct the matching code base (8K individually graded tasks have been contributed by the authors of said papers, grouped as rubric trees with weighting for the final grades). Benchmark is evaluated with an LLM judge (though I suspect some of it could be done automatically by constraining a bit the shape of the code asked for).

DSBench (2025) is a multimodal data analysis benchmark using Kaggle and ModelOff (financial data) samples. From the examples in Appendix it seems that questions from ModelOff are provided in a multiple choice setup, which likely makes the task easier, where the Kaggle tasks each have their own metric.

DABStep (2025) evaluates model on previously private (therefore uncontaminated) operational data analysis workloads using real life questions and data. All problems require multi step reasoning and varied document parsing, as well of course as specific data manipulation skills. It's a neat eval because it's hard and replicates actually useful real world use cases, and because each problem has a ground truth, so evaluation is unbiased and not too costly.

Assistant tasks test integrated capabilities in realistic scenarios, but they're either dynamic and read only, or static in environment which doesn't change. To evaluate adaptability and dynamic decision-making, we need environments that can "surprise" the model.

GAME BASED EVALUATIONS

Game-based benchmarks are very interesting for several reasons: they usually evaluate adaptability to a changing environment (contrary to most assistant tasks which are static), require long context reasoning, and last but not

least, are understandable by most people. However, they are not grounded in real life nor necessary reflecting good performance on actually useful use cases.

The most famous formal evaluation among these is probably ARC-AGI. The first version (2019) was made of puzzles grids in a sequence, where models had to find the last item of said sequence without explicit rules being provided. This benchmark is to me very reminiscent of logic-oriented IQ tests, and it was almost solved in 2024. A similar benchmark (extrapolation of rules) is Baba is AI (2024). The latest version of the bench, ARC-AGI3 (2025, ongoing), is still in development, and contains entire new games (requiring exploration, complex planning, memory management, ⋯) made specifically for the benchmark. It is still ongoing, and current best solutions on available problems are bruteforcing the games.

The community and model providers have explored a number of existing games with LLMs. Single player adventure games/RPGs like TextQuests (2025) or Pokemon (2024) (Twitch for Claude and Gemini for ex) require a combination of very long range planning to get objectives, which require adequante long context memory management, reasoning, and backtracking abilities. Same abilities are needed for single player survival games like Crafter (2021, Minecraft inspired). A number of single player game environments have been integrated into the Balrog (2024) benchmark.

Competitive bluffing games like Poker (2025), Mafia variations like Town of Salem (2025) and Werewolf (2025, here/there), or Among us are very interesting to test logic, reasoning, as well as deception abilities. Claude Opus 4 is for example incapable of winning Town of Salem as a vampire (deceptive role) but does well as a peasant (non deceptive role). Cooperative games like Hanabi can also be used to test adaptability and communication ability in a constrained environment.

What's also very neat about these is that they have a single and unambiguous pass/fail metric: did the LLM win the game or not? At the moment, if I were to use these to evaluate models I would probably look at TextQuests for abilities and Town of Salem for safety.

Beyond testing capabilities in controlled environments, people have explored the ultimate ungameable task: predicting the future.

FORECASTERS

In the last year, a new category of impossible to contaminate tasks emerged: forecasting. (I guess technically forecasting on the stock markets can be cheated on by some manipulation but hopefully we're not there yet in terms of financial incentives to mess up evals). They should require a combination of reasoning across sources to try to solve questions about not yet occuring events, but it's uncertain that these benchmarks are discriminative enough to have strong value, and they likely reinforce the "slot machine success" vibe of LLMs. (Is the performance on some events close to random because they are impossible to predict or because models are bad at it? In the other direction, if models are able to predict the event correctly, is the question too easy or too formulaic?)

FutureBench tests if models can predict future news-worthy events. It uses 2 sources: browsing and an LLM generating questions with a weekly time horizon, and user predictions from betting markets. All data is heavily filtered and cleaned before use. For now, models are barely better than random on human created bets, and succeed 3/4th of the time on model generated questions (likely easier).

FutureX is similar, but uses an array of specific websites (prediction parkets, government websites, general ranking websites and real time data platforms), then uses templates to generate questions about potential future events (`when will STOCK reach POINT?`). 500 questions are generated daily, with filtering of accidentally irrelevant questions.

A similar approach is used to generate questions in Arbitrage, the core difference being the time horizon: events there should be resolved in 2028.

In a similar vein, you'll also find arenas where LLMs are provided with money to actively trade on financial markets (like Alpha Arena or Trading Agents) - these experiments are less likely to give meaningful results, as, because of their costs, they tend to be run once per model only, so you get no statistical significance there.

RECOMMENDATIONS

🎯 TLDR

The landscape of evaluation has evolved with the jumps in capabilities, from testing isolated skills to measuring integrated performance in more realistic scenarios.

As of Nov 2025, I recommend using:

- Core capabilities (for model builders): Old capabilities evals for training, and for post training AIME26 when it will come out, GPQA, IFEval, SWE-Bench, a long range eval of your choice like HELMET, TauBench or BFCL if you're targetting tool use
- Core capabilities (for comparing models at inference): IFBench, HLE, MathArena, AiderBench and LiveCodeBench, MCP-Universe
- Long horizon tasks (for real-world performance): GAIA2, DABStep, SciCode, or domain specific evaluations for your use cases
- Games (for some extra fun in measuring robustness and adaptability): ARC-AGI3 when it's out, TextQuests, Town of Salem if you're interested in safety, or any other game you like which goes beyond Poker/Chess/Go.

The field is moving toward evaluations that test capability orchestration rather than isolated skills for actual use. This matches our goal of building models that "work well"—systems that can reliably combine core capabilities, tool use, with a good orchestration to solve actual problems.

> If you want to explore even more datasets, you'll find a big list of older interesting benchmarks here with my notes.

Understanding what's in there

No matter how you selected your initial datasets, the most important step is, and always will be, to look at the data, both what you have, what the model generates, and its scores. In the end, that's the only way you'll see if your evaluations are actually relevant for your specific use case.

You want to study the following.

DATA CREATION PROCESS

- Who created the actual samples? Ideally, you want dataset created by experts, then next tier is paid annotators, then crowdsourced, then synthetic, then MTurked. You also want to look for a data card, where you'll find

annotator demographics - this can be important to understand the dataset language diversity, or potential cultural bias.

- Were they all examined by other annotators or by the authors? You want to know if the inter-annotator score on samples is high (= are annotators in agreement?) and/or if the full dataset has been examined by the authors. This is especially important for datasets with the help of underpaid annotators who usually are not native speakers of your target language (think AWS Mechanical Turk), as you might otherwise find typos/grammatical errors/nonsensical answers.
- Were the annotators provided with clear data creation guidelines? In other words, is your dataset consistent?

SAMPLES INSPECTION

Take 50 random samples and manually inspect them; and I mean do it yourself, not "prompt an LLM to find unusual stuff in the data for you".

First, you want to check the content quality.

- Are the prompts clear and unambiguous?
- Are the answers correct? (*Eg: TriviaQA contains several gold answers (aliases field) per question, sometimes conflicting.*)
- Is information missing? (*Eg: MMLU references absent schematics in a number of questions.*)

It's important to keep in mind that it's not because a dataset is a standard that it's a good one - and this happens because most people skip this step.

Then, you want to check for relevance to your task. Are these questions the kind of questions you want to evaluate an LLM on? Are these examples relevant to your use case?

You might also want to check the samples consistency (especially if you're planning on using few shots or computing aggregated statistics): do all samples have the same number of choices if it's a multiple choice evaluation? Is the spacing consistent before and after the prompt? If your evaluation comes with an additional environment, ideally you want to use it to understand what gets called.

Lastly, you also want to quickly check how many samples are present there (to make sure results are statistically significant - 100 samples is usually a minimum for automatic benchmarks).

In the below viewer, for example, you can inspect the first samples of well known post-training benchmarks, collected by Lewis.

TASK AND METRICS

You want to check what metrics are used: are they automatic, functional, or using a model judge? The answer will change the cost of running evaluations for you, as well as the reproducibility and bias type. Best (but rarest) metrics are functional or based on rule based verifiers

So, you can't reproduce reported model scores?

Let's say you have read a recent tech report about a cool new model, and you want to reproduce their results on your machine… but you're not managing to? Let's explore why.

DIFFERENT CODE BASE

To reproduce evaluation scores to the decimal point, you first need to make sure you're using exactly the same code base as the paper you want to reproduce.

Usually, this means either using the evaluation default code as provided by the authors, or a standard implementation in a reference library like Eleuther's AI `lm_eval` or HuggingFace's `lighteval`. However, if the code source for evaluation is not provided, then, I'm sorry for you but it's unlikely that you'll be able to reproduce the results precisely.

If you want to easily understand what kind of discrepancies happen when using different implementations, you can explore this blog (⭐) we wrote with the eval team at HuggingFace. It studies the differences we observed between 3 common implementations of the MMLU evaluation (in `lm_eval`, `helm`, and in the original author implementation), and how they change model scores.

SUBTLE IMPLEMENTATION OR LOADING DIFFERENCE

We've observed that the following were easy things to mess up, even when using the same code base:

- Different random seeds.
  - Normally, inference is less affected by random seeds than training. However, they can still affect some CUDA operations (see the PyTorch page on reproducibility) and change predictions if you're using a non greedy generation strategy. They can also affect the prompt if you're using few-shots, and some pre or post-processing functions. -> A tiny change can result in a couple of points of difference.
- Actually different metrics. Metrics can be different in practice even if they share the same name. Some examples:
  - If the original implementation is a *log likelihood* `exact match` (computing the log probabilities of different possible answers), and you're using a *generative* `exact match` (only comparing the main greedy generation with the reference), you won't get the same scores.
  - We also saw, in evaluation code bases, a number of tasks which were defined as `exact match`, but were actually `prefix exact match` (comparing only the beginning of the generation with the reference), or `suffix exact match` (the opposite), or `quasi exact match` (exact match with a normalization). -> You therefore can't rely only on the metric name to determine what is happening, and need to look at the code.
- Different normalization.
  - To go back to our above `exact match` comparison example, in `lm_eval` v1, a number of tasks were simply named generative `exact match`: you would assume from this that the prediction is *compared as such* to a

reference. Looking at the code, the prediction would instead go through a normalization step (removing punctuation, homogenizing numbers, etc) before being compared to the reference. This will obviously change results quite a lot. (The `lm_eval` v2 now includes the normalization name in most metric names.) -> This is one of the easiest things to mess up, especially for tasks which require a lot of normalization/answer post processing, like math evaluations (where you want to extract the answer from a generated explanation).

> 🔧 Model loading affects reproducibility
>
> Four factors that change results even with identical code:
>
> - Hardware: PyTorch doesn't guarantee reproducibility across different GPUs/hardware
> - Inference library: transformers, vllm and sglang handle batching and matrix operations slightly differently as of 2025
> - Batch size: Different batch sizes = different results (you should fix the batch size for reproducibility, though careful about OOM errors)
> - Loading precision: Lower precision (especially quantized models vs floating point models) will change numerical results

DIFFERENT PROMPT

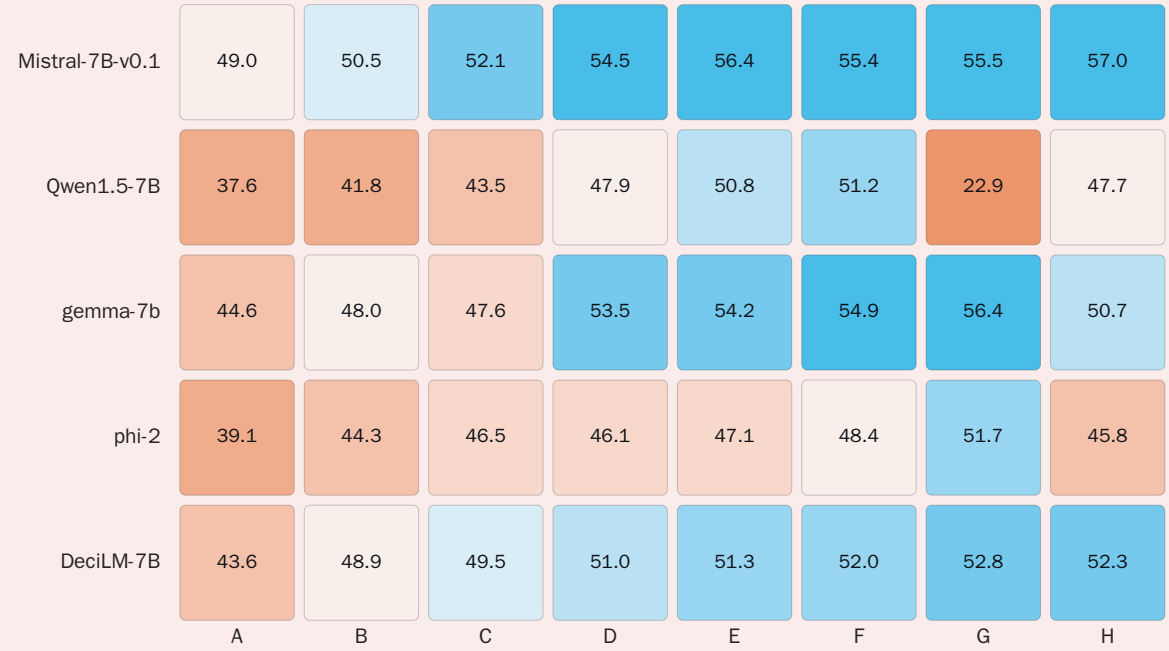
3 main things can come into play for prompt variation.

Prompt itself

The format you are using for the prompt can and will change scores wildly.

For example, for multichoice question answers, common formats include very simple variations (e.g. using `A` vs `A.` vs `A)` to introduce choices), which, while semantically equivalent (as they contain the exact same content) can still result in difference of *several points for the same model*.

> 📝 Prompt format sensitivity
>
> We did some experiments on this (you'll see up to a 7 points difference for the same model on the semantically equivalent prompts, the 5 rightmost columns), and a A paper observed similar results.
>
> Other example: Llama 3.1 models predicted correct MATH-Hard answers but scored poorly on the Open LLM Leaderboard, because they overfit to GSM8K's prompt format and couldn't adapt to the new one for this eval, despite it being provided in few shot examples.
>
> *Evaluation on MMLU subsets, acc_norm score (seed 0), in 5-shot.*



Prompt Formats:

A. ...? -> choice1/choice2/...

B. Q:...? A: -> choice1/choice2/...

C. Question: ...? Answer: -> choice1/choice2/...

D. Question: ...? Choices: ... Answer: -> choice1/choice2/...

E. Question: ...? Choices: A. ... Answer: -> choice1/choice2/...

F. Question: ...? Choices: (A) ... Answer: -> choice1/choice2/...

G. Question: ...? Choices: A. ... Answer: -> A/B/C/D

H. Question: ...? Choices: (A) Answer: -> (A)/(B)/(C)/(D)

This great paper⭐ also highlights a side effect of this: a number of models are now trained to overfit benchmark prompts and answer formats, to the cost of adaptation to other prompts at evaluation time.

Some tasks are also prefixed with a task prompt (eg: `The following questions are about <topic>`) - its presence or absence will also affect the scores.

System prompt and chat template

Chat models usually have been through instruction/preference training or fine-tuning. During this stage, they have learned to follow specific templates when inferring. For example, templates can require starting rounds of dialogue with a general prompt (called the `system prompt`) prefixed by specific tokens (usually `System:`). Said prompt is here to provide high-level instructions for the model, such as the contents of a persona, or general answering style instructions. Rounds of dialogue can also require adding prefix key words to text, such as `User` for queries and `Assistant` for answers.

When using few shot, you also need to select if you want examples to be provided multi-turn (mimicking user/assistant turns) or all at once (in a single user prompt).

Not following the chat template expected by the model at inference will kill its performance, as it will drive its output outside of the probability space it's been converging on.

Similarly, if you are using a reasoning model, you need to make sure whether you are comparing with or without thinking enabled.

Few-shots samples

Two things are easy to mess up with few-shot samples: the number of few-shot examples, which ones you are using, and their specific ordering

This is also a place where paying attention to the random seeds is important.

Parameters

For generative evaluations, parameters to pay attention to are making sure you are 1) using the same end of sentence token (you probably should not be using a default one for chat and reasoning models); 2) allowing your model to generate the same number of tokens for the evaluation (this is particularly crucial for reasoning models, which require a huge numbers of tokens in thinking mode); 3) if using sampling, that you are using the same seed/temperature parameters.

Selecting good benchmarks automatically for model training

In some cases, you don't want to "just" reproduce existing scores a posteriori, but you actually need to understand how well your model is training while it's happening. Evaluations you need then have different properties than evaluations for the final performance of models, as you need tasks which will provide good signal even when the model is not yet very good.

So the FineWeb team designed a method to select the best evaluations for pre-training ablations, across 9 languages - let's listen to their wise advice.

For these languages, we collected and implemented all available tasks that we could find, a total of 185 tasks. Then, we began task selection with two primary goals: ensuring evaluation diversity, and making sure each task provided a reliable signal during pre-training.

For evaluation diversity, we aimed to assess a broad range of model capabilities, including:

- Reading comprehension (RC): Understanding provided context and answering questions based on it.
- General knowledge (GK): Answering questions about facts from various fields without added context.
- Natural Language Understanding (NLU): Comprehending the semantics of provided input.
- Common-sense reasoning (RES): Demonstrating the ability to perform simple reasoning requiring embodied knowledge.
- Generative tasks: Ability to generate text in the target language without the "help" of multiple choice options.

We consider that tasks provide a reliable signal if they provide a dependable score. This means the score should be above the random baseline, increase as training progresses, show low variability across different seeds, and provide consistent model ranking at each training stepFor similar sized models trained with the same hyperparameters on the same amount of data..

To thoroughly examine the signal our tasks provide, we trained many 1.5B parameter models for each language, using 30B tokens from subsets of the supported languages of the five largest openly available multilingual web

datasets. These models were trained with the same hyperparameters and tokenizer. We then evaluated them at regular checkpoint intervals on the collected tasks (with no instruction and no system prompt in a 0-shot setting).

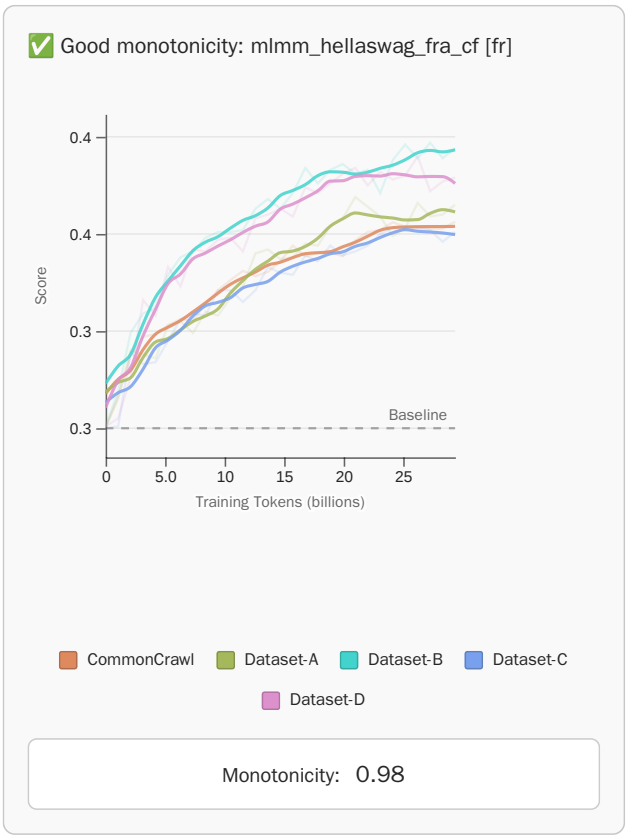
This process required multiple evaluation runs for each task due to iterations on its implementation, resulting in a total of 73 000 GPU hours consumed 🔥!

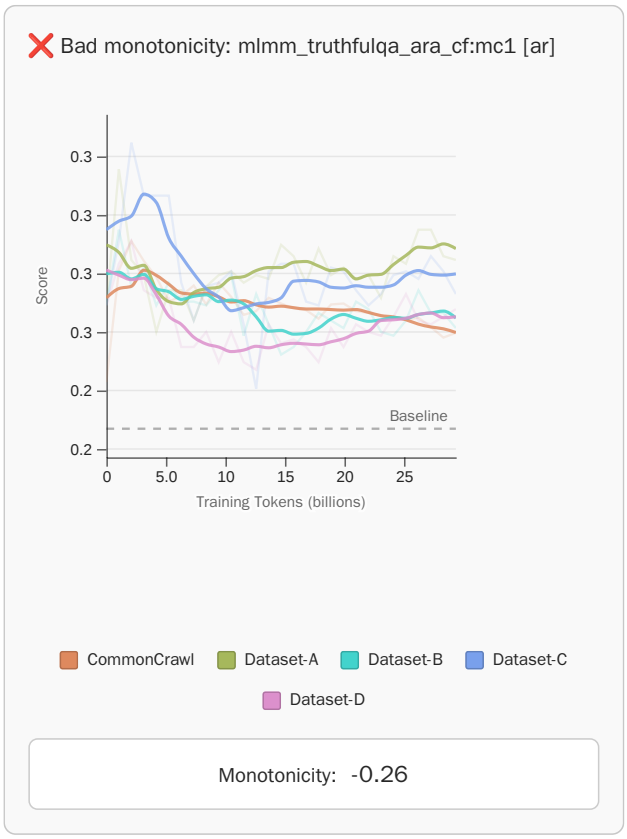
With 49 models trained we could finally define what a reliable signal means to us!

MONOTONICITY

One of our core requirements for a task is that it can be learned from training data and this learning can be gradually observed as the training progresses. Without this improvement through time, it's uncertain whether there will ever be an improvement in the future.

To measure this, we used the Spearman rank correlation to quantify the correlation between steps and score. Spearman rank correlation can capture monotonicity even when scores don't evolve linearly with the number of steps. We required each task to have at least an average correlation of 0.5 over all model training runs.





LOW NOISE

When comparing model performance on tasks, we need to consider whether differences are due to evaluation noise or genuine performance variations.

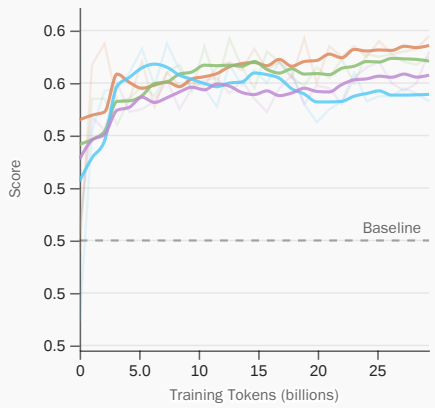
Noise can arise from the stochastic processes involved in model training, such as random token sampling, data shuffling, or model initialization (Madaan et al., 2024). To measure how sensitive each task is to this noise, we trained four additional models on our own monolingual corpora (unfiltered CommonCrawl data in each language) using different seeds.

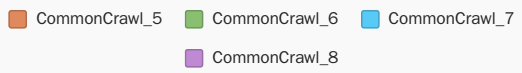
For each task, we computed:

1. First, a standard deviation of model scores for every step (approximately every 1B tokens), which we call the per-step-std.
2. Then, to obtain a global variability measurement, we averaged all the per-step-std values to get the avg-std over the full training. We assume this value is an upper-bound across model architectures and training datasets (as it was approximated by models trained on a “dirtier” dataset, therefore with higher variability).
3. Finally, we computed the signal-to-noise ratio (SNR) as the main metric for task variability. We calculate SNR as the mean score at 30B tokens of all runs divided by the avg-std. This metric measures how significant the overall score is relative to the score variations (noise).

We aimed for each task to have an SNR > 20. The only exception to this rule are generative tasks, which typically have relatively low SNR, but are still worth including as they provide insights into how the model behaves when prompted to generate unconstrained (without answer options). In a multilingual setting, this is particularly relevant as some models trained on multiple languages can exhibit high task scores but then suddenly reply in the wrong language for generative tasks!

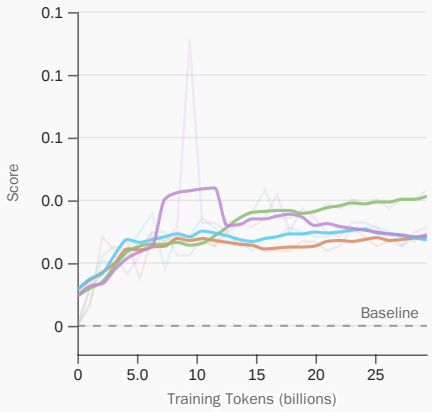
✅ Good SNR: xstory_cloze_tel_cf [te]

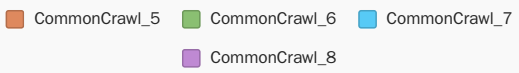




SNR: 87.87

❌ Bad SNR: tydiqa_tel [te]





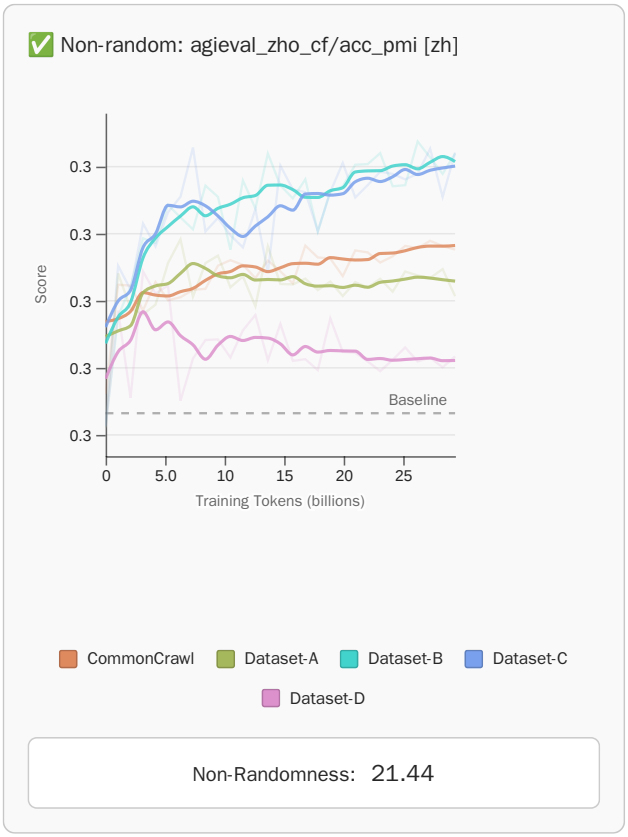
SNR: 6.40

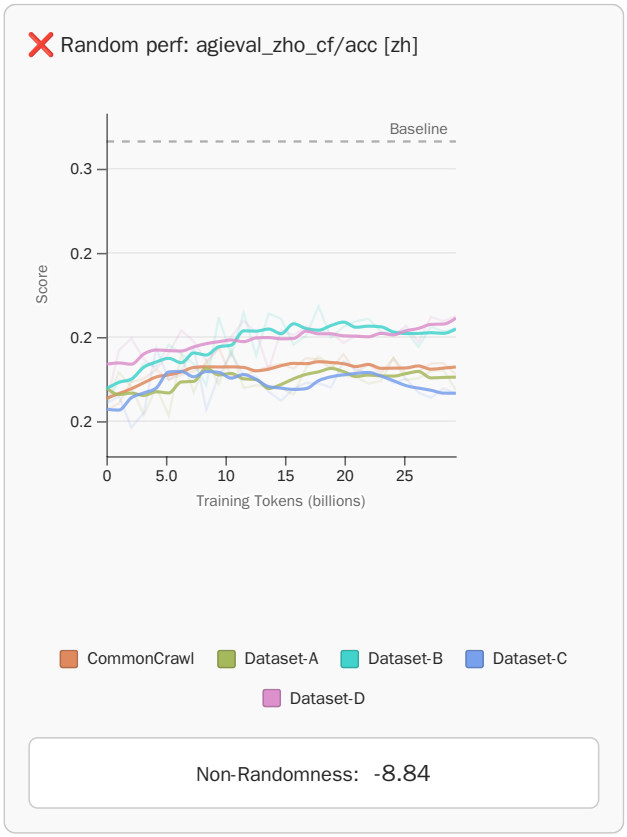
> Assuming model performance is normally distributed across different seeds, we want the benchmark-run performance to be at least 3 final-stds above the benchmark random baseline. This would mean that 99.85% of seed scores are above the random baseline (formally, benchmark-run performance - benchmark random baseline > 3 * final-std).

NON-RANDOM PERFORMANCE

Many model capabilities are acquired later in training, thus many tasks (especially harder ones, such as math-related ones) show baseline-level performance for an extended period. While these tasks are useful, they're not ideal for early pre-training evaluation, and we did not want to keep them for this setting.

We first computed the baseline random performance of the task (as the sum of 1/n_choices for all samples for multiple choice questions, and as zero for generative evaluations). Then we calculated the task's distance from the baseline as the maximum score across all models minus the baseline.





MODEL ORDERING CONSISTENCY

Let's not forget that the main goal of these evaluations is to compare models and datasets!

In the future, we want to use these evaluations to select the best datasets for full model pretraining. This means our tasks should rank datasets trained using very few tokens (we typically run data ablations on 30B tokens), in the same order as they would when trained for longer, after significantly more steps.

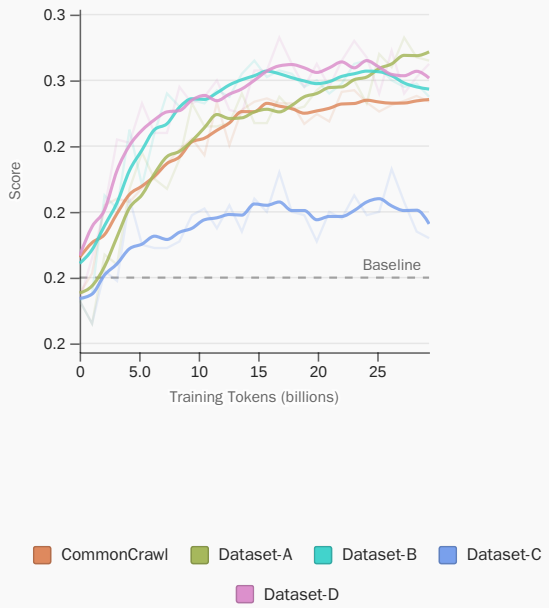
In other words, we would like tasks to have predictive capability regarding future performance during pre-training: if pre-training dataset A outperforms pre-training dataset B at 30 billion tokens, we would like this trend to continue at 300 billion tokens.

Proving this is inherently impossible, but there is a necessary preliminary condition that we can test for: for the results to be consistent at large scales, they must also first show consistency at smaller scales!

To measure this consistency in task ordering, we computed the average Kendall's Tau of models ranking between every two consecutive steps. We only considered steps starting after 15B tokens of pre-training, as we found orderings before the range incredibly noisy. A high value of this metric indicates that the ordering remains consistent as training progresses.

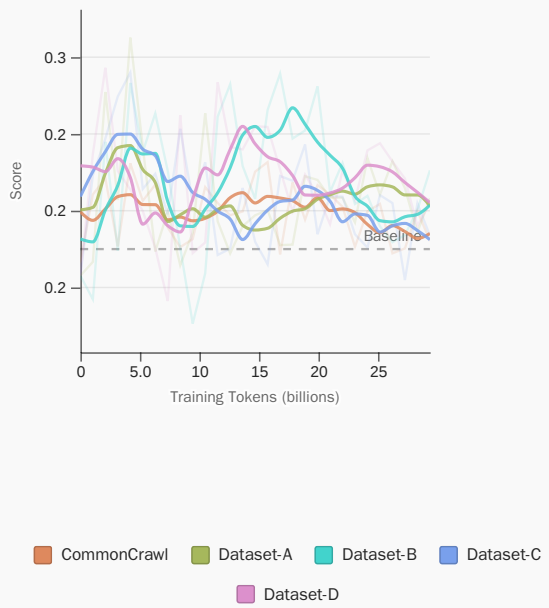
> We had no strict minimum value requirement for this property, instead using it to establish comparisons between tasks.

✅ Good ordering: xcsqa_ara_cf [ar]



Kendall's Tau: 0.83

❌ Bad ordering: thai_exams_tha_cf [th]



Kendall's Tau: 0.69

METRICS

As the targets in CF of multiple choice tasks are choices themselves, each target can have a different number of tokens, characters, and unconditional probability (probability of generating the choice without a context prefix).

> Measuring accuracy without normalization would have the models prefer answers with fewer tokens, for example.

To account for this, we consider the following accuracy variations:

- Accuracy : `acc` = $\arg\max_i(ln(P(a_i|q)))$
- Accuracy normalized over character length : `acc_char` = $\arg\max_i \frac{ln(P(a_i|q))}{num\_characters(a_i)}$
- Accuracy normalized over token length : `acc_token` = $\arg\max_i \frac{ln(P(a_i|q))}{num\_tokens(a_i)}$
- PMI Accuracy : `acc_pmi` = $\arg\max_i ln\frac{P(a_i|q)}{P(a_i|u)}$, where u = "Answer:"

Where a_i is the answer choice i, q is a question prompt and $P(a_i|q)$ is the probability of having a_i follow q. For more details see Gu et al., 2024 and Biderman et al., 2024.

> `acc_pmi` metric measures how much more likely a model is to predict A_i if provided with question context compared to if there was no context at all. This can be useful if the correct choice contains generally unlikely tokens, making the model less likely to choose such an answer.

For our generative tasks on the other hand, we used the following metrics:

- `prefix_match`: Exact match where only the prefix of the answer must match
- `f1`: F1 score computed over predicted/gold words extracted using a word tokenizer

For both generative metrics, minor preprocessing is applied to remove articles and punctuation, and lowercase the text.

Selecting the best evaluation metrics proved to be a challenging task. Not only is there no single metric that consistently outperforms the rest, but we often encountered situations where one metric had better monotonicity while another had a higher signal-to-noise ratio. In such cases, we typically made our decision based on the selected metric for tasks' implementation in a different language. We are aware that such hand-picking is often not possible and thus offer the following recommendations:

➡️ Multichoice Tasks

- We found base accuracy to perform well for tasks with answer options varying subtly (e.g. Yes/No/Also), particularly NLI tasks. In such cases, where the answer options are often each a single token, the base accuracy is advisable to use.
- While OLMES authors (Gu et al., 2024) recommends using PMI for tasks with unusual words, we found PMI to be highly effective for "difficult" reasoning and knowledge tasks like AGIEVAL or MMLU. In these cases, PMI provided the best results and was often the only metric delivering performance above random. That said, PMI was, on average, the weakest metric across all other tasks, while also being two times more expensive to compute. We therefore only recommend its use for complex reasoning and knowledge tasks.
- The metrics we found to be most reliable overall were length normalization metrics (token or character-based). However, the best choice was dependent on language, rather than being consistent for a given task. Due to that, we recommend using the maximum of acc_char and acc_token for the most reliable results.Note that acc_token is heavily tokenizer dependent. On our ablations all models were trained using the same tokenizer.

➡️ Generative Tasks

For generative metrics, the choice is clearer: we suggest using the F1 score unless exact matching is required, as in math-related tasks. F1 is generally less noisy and more resilient to small changes in the generations.

Creating your own evaluation

At this stage, you likely have a good idea of why people do evaluation, which benchmarks exist and are relevant for different model stages (training, inference of base and tuned models), but what if nothing exists for your specific use case?

This is precisely when you could want to create your own evaluation.

Dataset

USING EXISTING DATA

You can use existing datasets are are, and change the prompting or metrics associated (as has been done for older evaluations to adapt them to new prompting method), but you can also aggregate datasets.

Dataset aggregation is a good approach when you want to evaluate a specific capability that isn't well-covered by a single benchmark. Rather than starting from scratch, you can combine samples from multiple existing datasets to create a targeted evaluation suite. That's for examples what the authors of the "Measuring AGI" paper did recently to try to create a new "AGI evaluation" dataset.

When aggregating datasets, pay attention to whether

- they contain redundant data (most mathematics datasets are rewrites or aggregations of the same initial problems)
- you need balanced representation across sources (you might not want one dataset to dominate and skew your evaluation) - this will also determine whether to aggregate scores across all samples or per subset
- formats and difficulty levels are compatible (typically, if creating a unified dataset, beware of mixing up samples requiring sampling or not).

New research by EpochAI (2025) showcases how to best aggregate benchmarks together under a single framework to make the aggregated dataset harder overall and less prone to saturation.

USING HUMAN ANNOTATORS

I suggest reading Section 3 of this review of good practices in data annotation quality. If you want production level quality and have the means to implement all of these methods, go ahead!

Annotation Process

- Iterative Annotation
- Careful Data Selection
- Annotation Scheme
- Guideline Design
- Pilot Study
- Validation Step

Annotators

- Workforce Selection
- Qualification Test
- Annotator Training
- Annotator Debriefing
- Monetary Incentive

Quality Estimation

- Error Rate
- Control Questions
- Agreement

Quality Improvement

- Correction
- Updated Guidelines
- Filtering
- Annotator Feedback
- Annotator Deboarding

Adjudication

- Manual Curation
- Majority Voting
- Probabilistic Aggregation

Figure 1: Data annotation best practices recommended by the comprehensive review at aclanthology.org/2024.cl-3.1 - check out their Section 3.

However, important guidelines (no matter your project size) are the following, once you defined your task and scoring guidelines.

- Workforce selection, and if you can monetary incentive You likely want the people working on your task to:

1. obey some demographics. Some examples: be native speakers of the target language, have a higher education level, be experts in a specific domain, be diverse in their geographical origins, etc. Your needs will vary depending on your task.
2. produce high quality work. It's notably important now to add a way to check if answers are LLM-generated, and you'll need to filter some annotators out of your pool. *Imo, unless you're counting on highly motivated crowdsourced annotators, it's always better to pay your annotators correctly.*

- Guideline design Make sure to spend a lot of time really brainstorming your guidelines! That's one of the points on which we spent the most time for the GAIA dataset.
- Iterative annotation Be ready to try several rounds of annotations, as your annotators will misunderstand your guidelines (they are more ambiguous than you think)! Generating samples several times will allow your annotators to really converge on what you need.
    - Quality estimation and Manual curation You want to control answers (notably via inter-annotator agreement if you can get it) and do a final selection to keep only the highest quality/most relevant answers.

Specialized tools to build annotated high quality datasets like Argilla can also help you.

> Going further
>
> - ⭐ How to set up your own annotator platform in a couple minutes, by Moritz Laurer. A good read to get some hands on experience using open source tools (like Argilla and Hugging Face), and understanding better the dos and don'ts of human annotation at scale.
> - ⭐ A guide on annotation good practices. It's a review of all papers about human annotation dating from 2023, and it is very complete. Slightly dense, but very understandable.
> - Another guide on annotation good practices, by ScaleAI, specialised in human evaluations. Its a more lightweigth complement to the above document.
> - Assumptions and Challenges of Capturing Human Labels is a paper on how to look at source of annotator disagreement and mitigate them in practice

Practical tips and tricks

CREATING A DATASET SYNTHETICALLY

Using rule-based techniques

If your task allows, using procedurally generated benchmarks is a very good way to get a virtually infinite supply of samples and avoid contamination! They can generate unlimited fresh test cases algorithmically, while controlling difficulty and enabling automatic verification, ensuring models haven't seen examples during training.

For some examples, you can look at NPHardEval, DyVal, MuSR, BabiQA, ZebraLogic, IFEval, or GSMTemplate among others. NPHardEval generates complexity-grounded tasks like graph problems with automatic verification and monthly refreshes to reduce overfitting. MuSR creates complex reasoning instances like 1000-word murder mysteries using neurosymbolic generation. ZebraLogic algorithmically produces logic grid puzzles by generating

solutions and iteratively minimizing clues using SAT solvers. BabiQA simulates entities following successions of actions. IFEval tests instruction-following with 500+ prompts containing verifiable constraints like word counts that can be checked programmatically. GSM-Symbolic uses templates to generate diverse math questions.

Tasks which usually fit this paradigm test mathematical, logical, or coding abilities.

Creating synthetic data with models

If you want to create synthetic data, you usually start from a number of seed documents that will act as your ground truth. These can be internal and specific to your use cases, or available on the web and of high quality (like Wikipedia, Stack Overflow, ⋯). You'll then likely need to chunk your data into units of self contained meaning.

You'll then likely want a model to design questions from your data. For this, you will need to select a frontier model, and design a very good prompt asking the model to create use-case relevant questions from the provided data. It's better if you ask the model to provide the source on which it based its question.

You can also use seed prompts as examples to provide to an external model for it to write the prompt for your model to generate new questions, if you want to go full synthetic ^^

Once this is done, you can do an automatic validation by using a model from a different family line on your ground truth + questions + answer as a model judge.

> ⚠️ Always make sure that you're checking your data
>
> No matter how tempting it is to do everything automatically, you should always check your data at every step, to make sure your evaluations are qualitative. Evaluation is the name of the game and you need to use extremely good data.

MANAGING CONTAMINATION

In general, you should assume that a dataset publicly available on the internet is or will be contaminated.

Solutions to mitigate this include:

- providing a canary string in the evaluation set (like in BigBench): it is a specific character combination that model creators can look for in their training sets, which would indicate that it contains an evaluation
- providing evaluation sets in encrypted or gated forms so that they can't be parsed easily by web crawlers - therefore not ending up accidentally in training sets
- running dynamic benchmarks: benchmarks regularly updated through time so that models can't "learn the answers by heart" (but it makes datasets more costly)
- if you are running a benchmark, trying to detect contamination post-hoc (for example, by looking at the generation perplexity or designing adversarial versions of the prompts - however, no method is a foolproof contamination detection method)

However, it's not because a dataset is contaminated that it won't still be interesting and have signal during training, as we saw in the ablations section.

> A model which can only predict well on its training data (and has not latently learnt more high-level general patterns) is said to be overfitting. In less extreme cases, you still want to test if your model is able to generalize to data patterns which were not in the training set's distribution (for example, classify toxicity on stack overflow after having seen only toxicity on reddit).

Choosing a prompt

The prompt is going to define how much information is given to your model about the task, and how this information is presented to the model. It usually contains the following parts: an optional task prompt which introduces the task, and the format that the output should follow, attached context if needed (for example a source, an image), a problem prompt which is what you ask of the model, and optional options for multiple choice evaluations.

When defining your prompt, you need to be aware that even small changes in semantically equivalent prompts can make the results vary by quite a lot, and prompt formats might advantage or disadvantage specific models (See this section).

➡️ This can be mitigated by re-running the evaluation several times with prompt variations (but it can be costly), or simply running your evaluation once using a range of prompt formats allocated to different samples of equivalent difficulty.

➡️ You can also provide examples to your model to help it follow the expected format (using few-shot examples), and adding connector words helps this overall.

Choosing an inference method for your model

You'll need to choose what kind of inference method you need.

> Reminder about loglikelihood evaluations
>
> Using log-probabilities is good for multiple choice question answers (MCQA), to test model knowledge, or ability to disambiguate.
>
> - Pros:
> - Makes sure that all models have access to the correct answer
> - Provides a proxy for model "confidence" (and calibration)
> - Fast to evaluate, especially when we ask the model to predict only one token (A/B/C/D the indices of the choices, or Yes/No, etc).
> - Allow to get signal on small models' task performance
> - Cons:
> - Slightly over-scores small models which would have generated something outside of the range of available choices if given free rein.

- Some models favor specific choices based on the order in which they have been presented, which could lead to unrepresentative evaluations (unless you're re-running the evaluation n times by shuffling samples orders, which you should do for significance if you have the budget for!)

> 💡 Tip: an easy speed up for MCQA evaluations
>
> You can speed up your MCQA predictions by a lot if you make sure your model needs to predict only one token for the task.
>
> This way, instead of running your `number_of_choices` predictions (`context + choice 1`, `context + choice 2`, etc), you can simply run inference on `context` and compute the probability distribution on the full vocabulary (which will include all your one token choices) to get your logprobabilities of interest, and do this step in one pass.

> Reminder about generative evaluations
>
> Nowadays most evaluations are generative: using generations is very good for any task where you want to test fluency, reasoning, or the ability of your model to actually answer questions. It's also the most relevant way to evaluate reasoning models.
>
> - Pros:
>   - Should actually correlates with LLM ability to generate fluent text, will most of the time be what people are actually interested in
>   - The only way to evaluate both closed and open source models
> - Cons:
>   - Can be harder to score (see below)
>   - More expensive than log likelihood evaluations, especially if they include sampling or reasoning models

Scoring

If you are looking at log-probabilities, your metrics are going to be easy: you'll likely want to look at a variant of accuracy (how often the most likely choice is the best choice). It's important to normalize it by sequence length (either character, token, or pmi). You could also look at perplexity, recall, or f1 score.

If you're looking at generative evaluations, this is where it gets trickyyy, so the next chapter is specifically on this!

Evaluation's main challenge: Scoring free form text

Scoring free-form text is tricky because there are typically many different ways to express the same correct answer, making it hard to determine semantic equivalence through simple string matching, and output variations can make two semantically identical answers look completely different. Responses can be partially correct or contain a mix of accurate and inaccurate information. There can even be no single ground truth for the problem at hand, for example

for tasks requiring to judge coherence, helpfulness, and style, which are inherently subjective and context-dependent.

Automatically

When there is a ground truth, however, you can use automatic metrics, let's see how.

METRICS

Most ways to automatically compare a string of text to a reference are match based.

The easiest but least flexible match based metrics are exact matches of token sequences. While simple and unambiguous, they provide no partial credit - a prediction that's correct except for one word scores the same as one that's completely wrong.

The translation and summarisation fields have introduced automatic metrics which compare similarity through overlap of n-grams in sequences. BLEU (Bilingual Evaluation Understudy) measures n-gram overlap with reference translations and remains widely used despite having a length bias toward shorter translations and correlating poorly with humans at the sentence level (it notably won't work well for predictions which are semantically equivalent but written in a different fashion than the reference). ROUGE does a similar thing but focuses more on recall-oriented n-gram overlap. A simpler version of these is the TER (translation error rate), number of edits required to go from a prediction to the correct reference (similar to an edit distance). Lastly, you'll also find model-based metrics using embedding distances for similarity like BLEURT (it uses BERT-based learned representations trained on human judgments from WMT, providing better semantic understanding than n-gram methods, but requiring a model download and task-specific fine-tuning for optimal performance). I'm introducing here the most well known metrics, but all of these metrics have variations and extensions, among which CorpusBLEU, GLEU, MAUVE, METEOR, to cite a few.

REF: My cat loves doing model evaluation and testing benchmarks

PRED: My cat enjoys model evaluation and testing models

Exact Match

0.0

Binary: 1 or 0

✗ Strings differ

Most strict metric - no partial credit

Translation Error Rate

0.333

Edit distance normalized

3 edits / 9 words = 0.333

Edit operations:

- Insert "loves"
- Replace "enjoys" → "doing"
- Replace "models" → "benchmarks"

Lower is better (0 = identical)

BLEURT

0.318

Semantic similarity

BLEURT uses BERT embeddings learned from real text.

BLEU

0.523

N-gram precision-based

1-gram: 6/8 (75%)

my | cat | model | +3 more

2-gram: 4/7 (57%)

my cat | model evaluation | evaluation and | +1 more

3-gram: 2/6 (33%)

model evaluation and | evaluation and testing

ROUGE-1

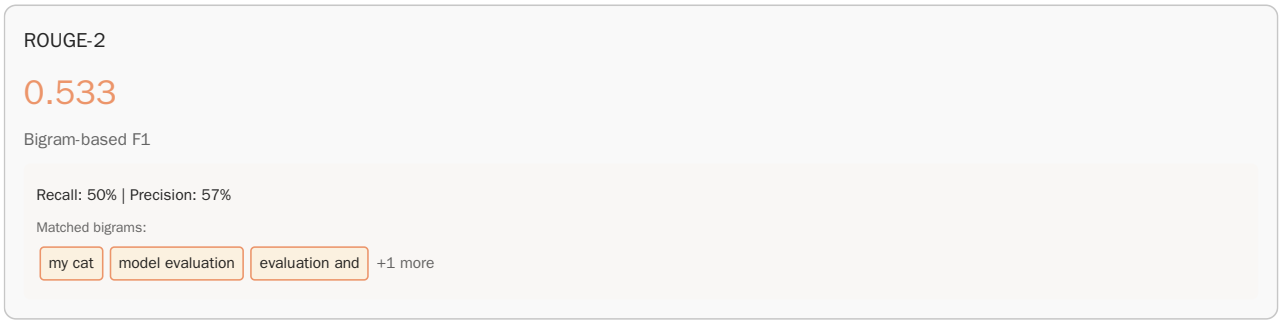
0.706

Unigram-based F1

Recall: 67% | Precision: 75%

Matched unigrams:

my | cat | model | evaluation | and | +1 more



Once you have an accuracy score per sample, you can aggregate it across your whole set in several ways. In general, people average their results, but you can do more complex things depending on your needs. (Some metrics already come with an aggregation, like CorpusBLEU).

If your score is binary, look at the precision (critical when false positives are costly), recall (critical when missing positives is costly), F1 score (balances precision and recall, good for imbalanced data), or MCC (Matthews Correlation Coefficient, which works well with imbalanced datasets by considering all confusion matrix elements). If your score is continuous (less likely though), you can use mean squared error (penalizes large errors but heavily weights outliers) or mean absolute error (more balanced than MSE).

More generally, when picking your metric and its aggregation, you need to keep in mind what your task is really about. For some domains (ex: medical, chatbots with public interaction), you don't want to measure the average performance, but need a way to evaluate the worst performance you'll get (on medical quality of output, on toxicity, etc).

> To go further
>
> - This blog covers some of the challenges of evaluating LLMs.
> - If you're looking for metrics, you'll also find a good list with description, score ranges and use cases in this organisation.

> Pros and cons of using automated metrics
>
> Automated benchmarks have the following advantages:
>
> - Consistency and reproducibility: You can run the same automated benchmark 10 times on the same model and you'll get the same results (baring variations in hardware or inherent model randomness). This means that you can easily create fair rankings of models for a given task.
> - Scale at limited cost: They are one of the cheapest way to evaluate models at the moment.
> - Understandability: Most automated metrics are very understandable.
>
> However, they also present have a reduced use on more complex tasks: an automatic metric either requires you to have a perfect, unique and unambiguous reference/gold, like for tasks where performance is easy to define and assess (for example, classification of toxicity, knowledge questions with a single answer). More complex capabilities, on the other hand, are harder to decompose into a single and simple answer.

NORMALIZATION

Normalization means changing a string of characters to have it fit a specific reference format. For example, when comparing a model prediction to a reference, you usually don't want to penalize extra spacing in the prediction, or added punctuation or capitalisation. That's why you normalize your prediction.

They are vital for specific tasks, such as math evaluations, where you want to extract an equation from a longer prediction, and compare it to a reference. In the below table, we make a list of some issues we saw happening when extracting predictions from model outputs using SymPy naively for the MATH dataset, and how Math-Verify, a specific math parser, solved these.

📄 Example	❗ Issue	✅ Math-Verify	🛑 Naive Approach
Therefore, the perimeter of one of these triangles is $14+7\sqrt{2}$ inches, expressed in simplest radical form.	Failed extraction	`7*sqrt(2) + 14`	None
Therefore, the sum of the infinite geometric series is (\frac79).	Failed extraction	`7/9`	None
The final answer is $2x+4y+z-19=0$. I hope it is correct.	Partial parse of parametric eq	Eq(2*x* + 4*y* + z - 19, 0)	0
(23)	Failed extraction due to latex borders	`23`	None
((- \infty, -14) \cup (-3, \infty)).	Failed extraction due to interval	Union(Interval.open(-oo, -14), Interval.open(-3, oo))	None
100%	Failed extraction due to invalid symbol	`1`	None
1/3 == 0.333333	No rounding support	True	False
sqrt(1/2)*7 == sqrt(0.5)*7	No numerical evaluation support	True	False

Normalizations can easily be unfair if not designed well, but overall they still help provide signal at the task level.

They are also be important for evaluation of predictions generated with chain of thought, or reasoning, as you'll need to remove the reasoning trace (which is not part of the final answer) from the output to get the actual answer.

SAMPLING

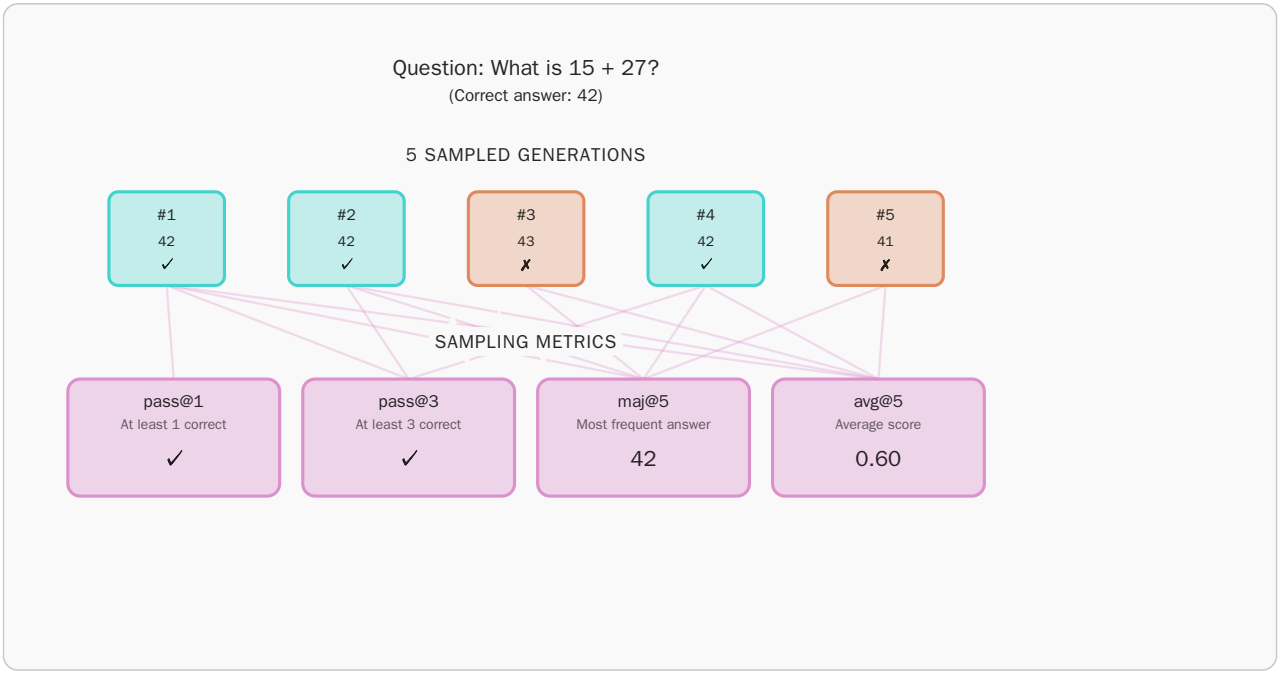
When models generate outputs, sampling multiple times and aggregating results can provide a more robust signal than a single greedy generation. This is particularly important for complex reasoning tasks where models may arrive at correct answers through different paths.

Common sampling-based metrics are:

- pass@k over n: Given n generated samples, measures whether at least k passes the test.
- maj@n (majority voting): Sample n generations and take the most frequent answer. This helps filter out spurious outputs and works particularly well when the model's correct reasoning path is more consistent than its errors. Commonly used for math and reasoning tasks.

- cot@n (chain-of-thought sampling): Sample n reasoning traces and evaluate them. Can be combined with majority voting or a pass@k (sample n reasoning chains, extract final answers, take majority or a threshold).
- avg@n (stable average score): Average the scores across n samples. It's a more stable estimator of performance than using "best" or "most common" case.

Sampling metrics comparison



When you use sampling evaluations, make sure to always report all sampling parameters (temperature, top-p, k value) as they significantly affect results.

> When can you use sampling and when shouldn't you?
>
> - For training evaluation/ablations: ❌ Generally avoid sampling metrics as they're expensive and add variance. Stick to greedy decoding with a fixed seed.
> - For post-training evaluation: ✅ Sampling metrics can reveal capabilities that greedy decoding misses (especially for more complex tasks requiring reasoning, math or code).
> - At inference: ✅ These metrics help estimate how much improvement you can get from sampling multiple times at inference. It's particularly cool when you want to study how far you can push small models with test time compute.
>
> However, keep in mind that sampling k times multiplies your evaluation cost by k. For expensive models or large datasets, this adds up very quickly!

FUNCTIONAL SCORERS

Instead of comparing generated text to a reference through fuzzy string matching, functional testing evaluates whether outputs satisfy specific verifiable constraints. This approach is extremely promising because it's more flexible and allows "infinite" updates of the test case through rule-based generation (which reduces overfitting).

IFEval and IFBench are excellent examples of this approach for instruction following evaluation. Rather than asking "does this text match a reference answer?", they ask "does this text satisfy formatting constraints given in the instructions?"

For instance, instructions might specify:

- *"Include exactly 3 bullet points"* → verify the output contains exactly 3 bullets
- *"Capitalize only the first sentence"* → parse and check capitalization patterns
- *"Use the word 'algorithm' at least twice"* → count word occurrences
- *"Your response must be in JSON format with keys 'answer' and 'reasoning'"* → validate JSON structure

Each constraint can be checked with a specific rule-based verifier, making these evaluations more unambiguous, interpretable, fast, and considerably less costly than using models as judges.

This functional approach works particularly well for instruction following, but requires creativity to extend to other text properties. The key is identifying aspects of text that can be verified programmatically rather than through semantic comparison.

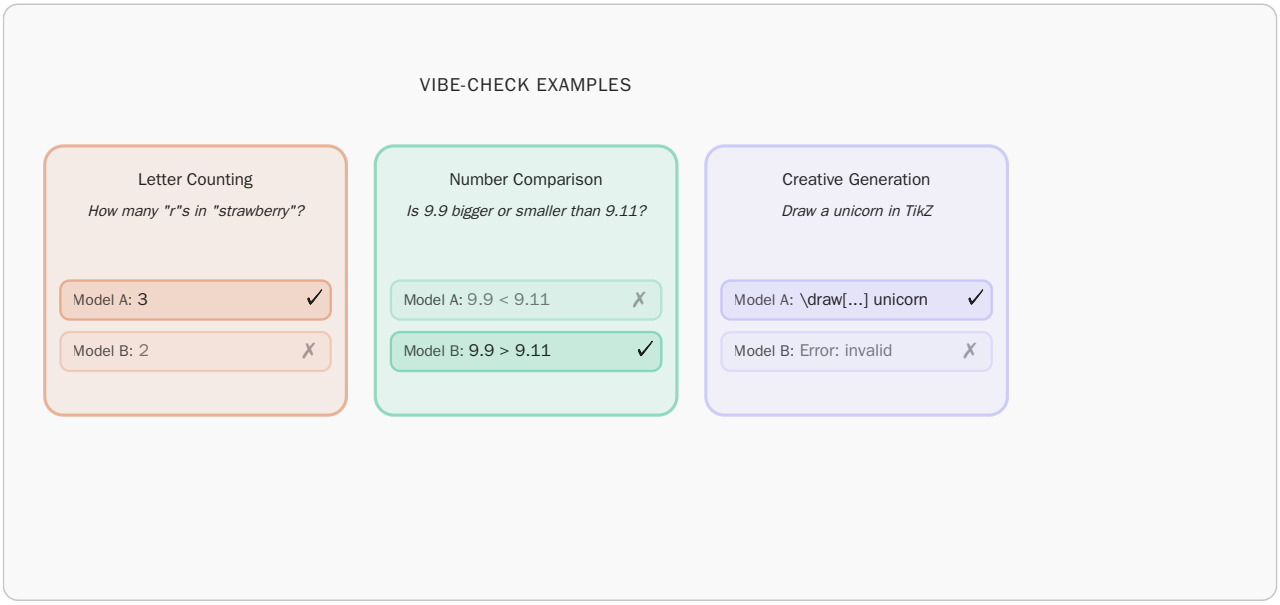
With humans

Human evaluation is simply asking humans to score predictions.

Human evaluation is very interesting, because of its flexibility (if you define clearly enough what you are evaluating, you can get scores for about anything!), inherent un-contamination (if humans write new questions to test your system, they should not be present in your training data, hopefully), and good correlation with human preference for obvious reasons.

Different approaches exist to evaluate models with humans in the loop.

Vibe-checks is the name given to manual evaluations done by individual members of the community, usually on undisclosed prompts, to get an overall "feeling" of how well models perform on their use cases of preference. (I've also seen the term "canary-testing" used for this, in reference to high signal canary in a coalmine approach). Said use cases can be anything from the most exciting to the most mundate - to cite some I've seen on Reddit, they covered legal questions in German, coding, tool use, quality of erotica written, etc. Often shared on forums or social media, they mostly constitute anecdotal evidence, and tend to be highly sensitive to confirmation bias (in other words, people tend to find what they look for).



Using community feedback to establish massive model rankings is what we call an arena. A well known example of this is the LMSYS chatbot arena, where community users are asked to chat with models until they find one is better than the other. Votes are then aggregated in an Elo ranking (a ranking of matches) to select which model is "the best". The obvious problem of such an approach is the high subjectivity - it's hard to enforce a consistent grading from many community members using broad guidelines, especially since annotators preferences tend to be culturally bound (with different people favoring different discussion topics, for example). One can hope that this effect is smoothed over by the sheer scale of the votes, through a "wisdom of the crowd" effect (this effect was found by a statistician named Galton, who observed that individual answers trying to estimate a numerical value, like the weight of a hog, could be modeled as a probability distribution centered around the actual answer).

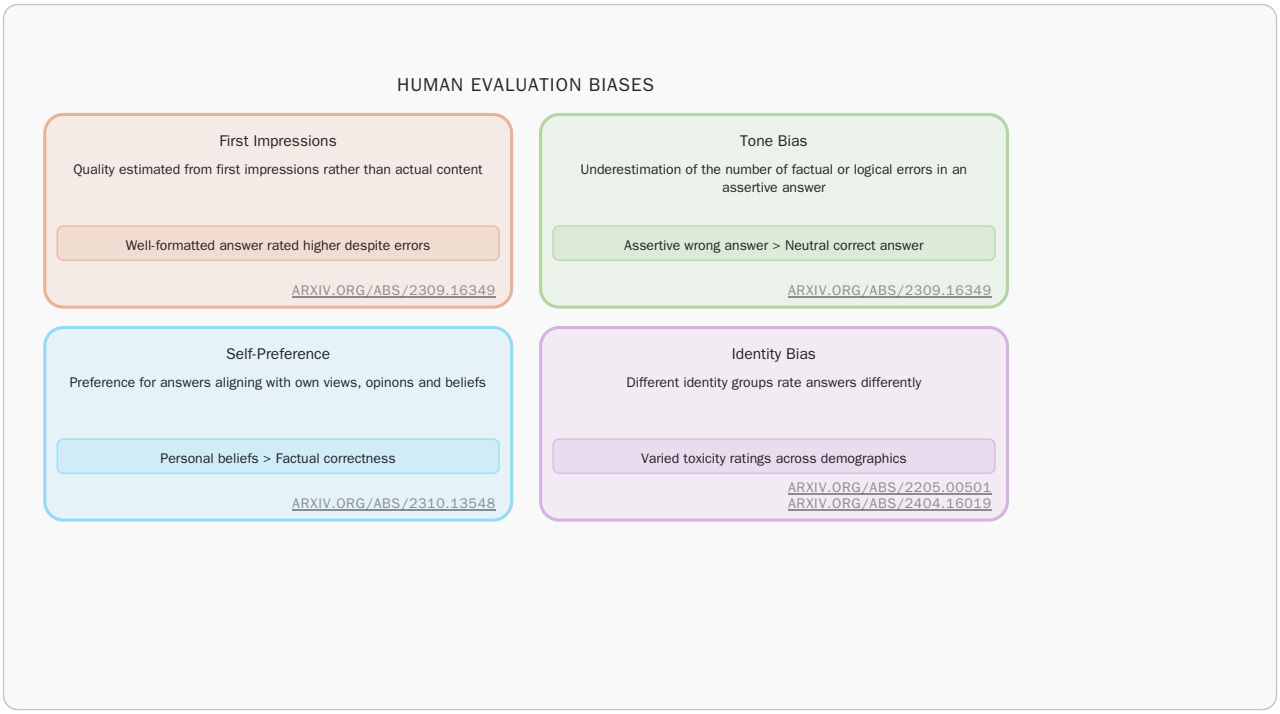
The last approach is systematic annotations, where you provide extremely specific guidelines to paid selected annotators, in order to remove as much as the subjectivity bias as possible (this is the approach used by most data annotation companies). However, it can get extremely expensive fast, as you have to keep on doing evaluations in a continuous and non automatic manner for every new model you want to evaluate, and it can still fall prey to human bias (this study showed that people with different identities tend to rate model answer toxicity very differently).

Vibe-checks are a particularly good starting point for your own use cases, as you'll be testing the model on what's relevant to you. Pros of casual human evaluations are that they are cheap and allow to discover fun edge cases since you leverage user's creativity in a mostly unbounded manner, you can discover interesting edge cases. However, they can be prone to blind spots.

Once you want to scale to more systematic evaluation with paid annotators, you'll find that there are 3 main ways to do so. If you don't have a dataset, but want to explore a set of capabilities, you provide humans with a task and scoring guidelines (e.g *Try to make both these model output toxic language; a model gets 0 if it was toxic, 1 if it was not.*), and access to one (or several) model(s) that they can interact with, then ask them to provide their scores and reasoning. If you already have a dataset (eg: a set of *prompts that you want your model to never answer*, for example for safety purposes), you preprompt your model with them, and provide the prompt, output and scoring guidelines to humans. If you already have a dataset and scores, you can ask humans to review your evaluation method by doing error annotation (*it can also be used as a scoring system in the above category*). It's a very important step of testing new evaluation system, but it technically falls under evaluating an evaluation, so it's slightly out of scope here.

Pros of systematic human evaluations, especially with paid annotators, are that you're getting high quality and private data adapted to your use case (especially if you rely on in house annotators), which are mostly explainable (scores obtained by the models will be explainable by the humans who gave them). However, it's more costly (especially as you'll most likely need rounds of annotations to adapt your guidelines) and does not scale well.

Overall, however, human evaluation has a number of well known biases, based first impressions, tone, alignement with annotators value, etc, see the figure below.



These biases are not unexpected, but they must be taken into account: not all use cases should rely on using cheap human annotators - any task requiring factuality (such as code writing, evaluation of model knowledge, etc) should include another, more robust, type of evaluation to complete the benchmark (experts, automatic metrics if applicable, etc).

With judge models

To mitigate the cost of human annotators, some people have looked into using models or derived artifacts (preferably aligned with human preferences) to evaluate models' outputs.

Judge models are simply neural network used to evaluate the output of other neural networks. In most cases, they evaluate text generations.

Two approaches exist for grading: using generalist, high capability models or using small specialist models trained specifically to discriminate from preference data (think "spam filter", but for toxicity for example). In the former case, when using an LLM as a judge, you give it a prompt to explain how to score models (ex: `Score the fluency from 0 to 5, 0 being completely un-understandable, ...`).

Model as judges allow to score text on complex and nuanced properties. For example, an exact match between a prediction and reference can allow you to test if a model predicted the correct fact or number, but assessing more open-ended empirical capabilities (like fluency, poetry quality, or faithfulness to an input) requires more complex evaluators.

They are used on 3 main tasks:

- *Scoring a model generation*, on a provided scale, to assess a property of the text (fluency, toxicity, coherence, persuasiveness, etc).
- *Pairwise scoring*: comparing a pair model outputs to pick the best text with respect to a given property
- *Computing the similarity* between a model output and a reference

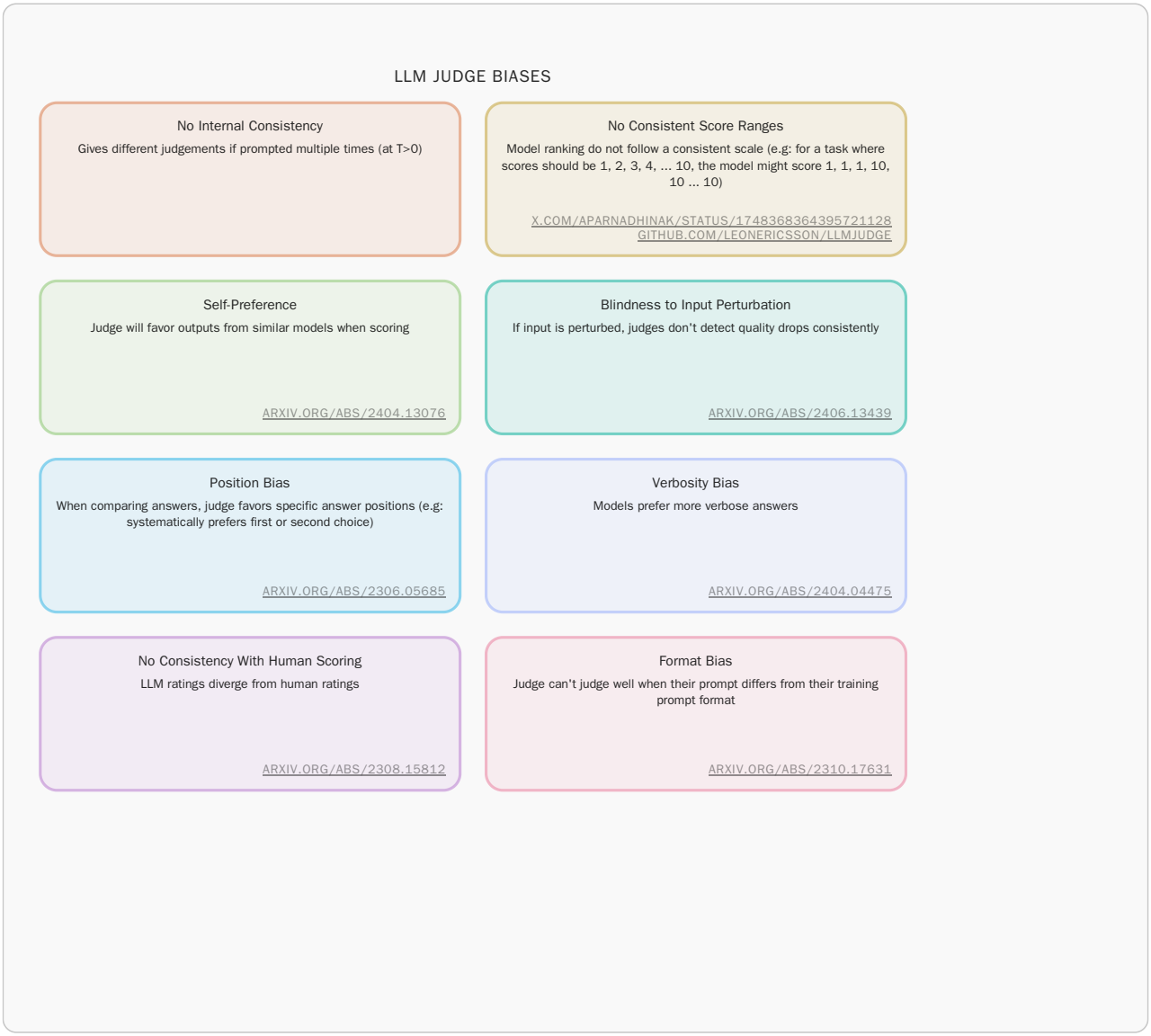
PROS AND CONS OF USING JUDGE-LLMS

People in favor of judge LLMs have been claiming they provide better:

- Objectivity when compared to humans: They automate empirical judgments in an objective and reproducible manner (theoretically - in my opinion, they add more subtle bias than they are worth)
- Scale and reproducibility: They are more scalable than human annotators, which allows to reproduce scoring on large amounts of data (if you control for temperature).
- Cost: They are cheap to instantiate, as they don't require to train a new model, and can just rely on good prompting and an existing high quality LLM. They are also cheaper than paying actual human annotators (capitalism⋯).

In my opinion, using LLM judges correctly is extremely tricky, and it's easy to be deceived for critical use cases:

- LLM as judges seem objective, but they have many hidden biases that can be harder to detect than the ones in humans, since we're not as actively looking for them (see below). Besides, there are ways to reduce human bias by designing survey questions in specific and statistically robust ways (which has been studied in sociology for about a century), where LLM-prompting is not as robust yet. Using LLMs to evaluate LLMs has been compared to creating an echo-chamber effect, by reinforcing biases subtly.
- They are indeed scalable, but contribute to creating massive amounts of data which themselves need to be examined to ensure their quality (for example, you can improve the quality of LLM-judges by asking them to generate a thinking trace, or reasoning around their data, which makes even more new artificial data to analyse)
- They are indeed cheap to instantiate, but are not as good as paying actual expert human annotators for your specific use cases.



This section is therefore a bit long, because you need to be well aware of the limitations of using model as judges: a lot of people are blindly jumping into using them because they seem easier than actually working with humans or designing new metrics, but then end up with uninsterpretable data with tricky bias to extract.

My main personal gripe with using models as judges is that they introduce very subtle and un-interpretable bias in the answer selection. I feel that, much like when crossbreeding too much in genetics studies, you end up with dysfunctional animals or plants, by using LLMs to select and train LLMs, we are just as likely to introduce minute changes that will have bigger repercussions a couple generations down the line. I believe this type of bias is less likely to occur in smaller and more specialized models as judges (such as toxicity classifiers), but this remains to be rigorously tested and proven.

> Getting started with an LLM judge
>
> If you want to give it a go, I suggest first reading this very good guide on how to setup your first LLM as judge!
>
> You can also try the distilabel library, which allows you to generate synthetic data and update it using LLMs. They have a nice tutorial applying the methodology of the Ultrafeedback paper as well as a tutorial on benchmarking implementing the Arena Hard benchmark.

GETTING A JUDGE-MODEL

When using an existing LLM, you can go for generalist, high capability models, small specialist models trained specifically to discriminate from preference data, or training your own.

Using a generalist LLM

With the introduction of more capable LLMs (such as ChatGPT), some researchers started exploring using big models as judges.

> ⚖️ Closed vs open source judge models
>
> Closed source models (Claude, GPT-o) tradeoffs:
>
> Disadvantages:
>
> - Non-reproducible: Models can change without notice via API updates
> - Black box: Un-interpretable decision-making
> - Privacy risks: Data sent to third parties, potential leakage
>
> Advantages:
>
> - Easy access without local setup or hardware requirements
>
> Open source models are closing the gap while solving reproducibility and interpretability issues. Models like DeepSeek R1, gpt-oss, and the recent Qwen models are now competitive alternatives.

You'll find a good cost analysis of model providers here if you need help picking one.

Using a tiny specialized LLM judge model

You can also make the choice to use tiny specialized LLM judges. With often a couple billion parameters, they can run locally on most recent consumer hardware, while being trained from scratch or fine-tuned using instruction data. You often need to follow their specific prompt formats.

Some existing models as of 2024 were Flow-Judge-v0.1 (weights), 3.8B parameters, a Phi-3.5-mini-instruct fine-tuned on a synthetic preference dataset, Prometheus (weights, paper), 13B parameters, a model trained from scratch on synthetic preference dataset, and JudgeLM (paper), 7B to 33B parameters, models trained from scratch on synthetic preference datasets generated with a variety of models. Newer alternatives surely exist!

Training your own You can also make the choice to train or fine-tune your own LLM-as-judge. (I would avoid doing this, unless you are on a very niche domain).

If you go in that direction, you'll first need to gather preference data for your task of interest, which can come

- From existing human preference datasets
- From model generated preference data (which you can generate following the above tiny-model judges papers data sections, or get directly, for example from the Prometheus preference and feedback collections).

Then you need to decide whether to start from a small model to train from scratch, or from an existing model, that you can distill into a new smaller model, or quantize, then fine-tune (using peft or adapter weights if the model is big

and your training compute low) using the above data.

DESIGNING YOUR EVALUATION PROMPT

Once you've selected your model, you need to define what is the best possible prompt for your task.

> 📝 Prompt design guidelines
>
> Provide a clear description of the task at hand:
>
> - *Your task is to do X.*
> - *You will be provided with Y.*
>
> Provide clear instructions on the evaluation criteria, including a detailed scoring system if needed:
>
> - *You should evaluate property Z on a scale of 1 - 5, where 1 means …*
> - *You should evaluate if property Z is present in the sample Y. Property Z is present if …*
>
> Provide some additional "reasoning" evaluation steps:
>
> - *To judge this task, you must first make sure to read sample Y carefully to identify …, then …*
>
> Specify the desired output format (adding fields will help consistency)
>
> - *Your answer should be provided in JSON, with the following format {"Score": Your score, "Reasoning": The reasoning which led you to this score}*

You can and should take inspiration from MixEval or MTBench prompt templates.

> ⚠️ To remember when doing model as judge
>
> Pairwise comparison correlates better with human preference than scoring, and is more robust generally.
>
> If you really want a score, use an integer scale make sure you provide a detailed explanation for what each score represents, or an additive prompt (*provide 1 point for this characteristic of the answer, 1 additional point if …* etc)
>
> Using one prompt per capability to score tends to give better and more robust results

You can also improve accuracy using the following, possibly more costly, techniques:

- Few shot examples: like in many other tasks, if you provide examples it can help its reasoning. However, this adds to your context length.
- Reference: you can also enhance your prompt with a reference if present, which increases accuracy
- CoT: improves accuracy for older gen models, if you ask the model to output its chain of thought before the score (also observed here)
- Multiturn analysis: can improve factual error detection
- Using a jury (many judges, where you pick an aggregate of the answers): gives better results than using a single model. It can be made considerably less costly by leveraging many smaller models instead of one big expensive

model. You can also experiment with using one model with variations on temperature

- Surprisingly, the community has found that adding stakes to the prompts (`answer correctly and you'll get a kitten`) can increase correctness. Your mileage may vary on this one, adapt to your needs.

If you are working on critical tasks (medical domain for example), make sure to use methodologies transferred from the humanities, and 1) compute inter-annotator agreement metrics to make sure your evaluators are as unbiased as possible, 2) Use proper survey design methodology when creating your scoring grid to mitigate bias. However, most people don't really want a reproducible and high quality unbiased eval, and will be happy with quick and dirty evaluation through OK-ish prompts. (Which is an OK situation to be in! Just depends on the consequences attached).

EVALUATING YOUR EVALUATOR

Before using a judge-LLM in production or at scale, you want to evaluate its quality for your task, to make sure its scores are actually relevant and useful for you.

> This will be easier to do if it predicts binary outputs, because you'll be able to interpretable classification metrics (accuracy/recall/precision). If it predicts scores on a scale, it will be much harder to estimate the quality of the correlation with a reference. Models are notoriously bad at predicting on a scale.

So, once you have selected your model judge and its prompt, you'll need to do the following.

1. Pick your baseline You'll need to compare your evaluator judgments to a baseline: it can be human annotations, the output of another judge model that you know is qualitative on your task, a gold truth, itself with another prompt, etc.

> 🎯 Quality over quantity for baseline
>
> You don't need many baseline examples (50 can suffice), but they must be:
>
> - Representative: Cover the full range of your task
> - Discriminative: Include edge cases and challenging examples
> - High quality: Use the best reference data you can obtain

2. Pick your metric Your metric will be used to compare your judge's evaluations with your reference.

In general, this comparison is considerably easier to do if your model is predicting binary classes or doing pairwise comparison, as you'll be able to compute accuracy (for pairwise comparison), or precision and recall (for binary classes), which are all very easy to interpret metrics.

Comparing the correlation of scores with human or model scoring will be harder to do. To understand why in more detail, I advise you to read this cool blog section on the topic.

In general, if you're a bit lost about what metrics to pick when (in terms of models, metrics, ⋯), you can also look at this interesting graph from the same above blog ⭐.

3. Evaluate your evaluator For this step, you simply need to use your model and its prompt to evaluate your test samples! Then, once you get the evaluations, use your above metric and reference to compute a score for your evaluations.

You need to decide what your threshold for acceptance is. Depending on how hard your task is, you can aim for 80% to 95% accuracy, if you're doing pairwise comparison. Regarding correlations (if you're using scores), people in the literature tend to seem happy with 0.8 Pearson correlation with a reference. However, I've seen some papers declare that 0.3 indicates a good correlation with human annotators (^^") so ymmv.

TIPS AND TRICKS

> ⚠️ Mitigating well known biases of LLM as judges
>
> We discussed in this section's intro a number of LLM judges biases. Let's see how you should try to mitigate them.
>
> Lack of internal consistency: ➡️ You can mitigate this by doing self-consistency prompting of your judge, prompting it multiple times and keeping the majority output
>
> Self-preference: ➡️ You can mitigate this by using a jury
>
> Blindness to input perturbation: ➡️ asking the model to explain its reasoning before providing a score ➡️ or providing a coherent grading scale in the prompt.
>
> Position-bias: ➡️ switching answer positions randomly ➡️ computing the log-probabilities of all possible choices to get a normalized answer
>
> Verbosity-bias (or length-bias): ➡️ You can mitigate this by accounting for the answer difference in length
>
> Format bias: ➡️ You can mitigate this by paying attention to the training prompt format (if the model was instruction tuned) and ensuring you follow it.

Picking correct tasks for an LLM judge

LLM evaluators:

- are bad at identifying hallucinations in general, particularly what are called partial hallucinations (which look close to the ground truth but are actually slightly different) (see this and this)
- have a low to OK-ish correlation with human annotators on summarization (here too), faithfulness, and are not consistently correlated with human judgement more broadly against a scope of tasks

WHAT ABOUT REWARD MODELS?

Reward models learn to predict a score from human annotations for given prompt/completion pairs. The end goal is for them to do predictions aligned with human preference. Once trained, these models can then be used to improve other models, by acting as a a reward function which is a proxy for human judgment.

The most common type of reward model is the Bradley-Terry model, which outputs a single pairwise score, following:

$$p(\text{completion b is better than completion a}) = \text{sigmoid}(\text{score}_b - \text{score}_a)$$

This model is trained using only pairwise comparisons of completions, which are easier to collect than scores, but can only compare several completions for one prompt, and not completions across prompts.

Other models have expanded on this approach to predict a more nuanced probability that a completion is better than the other one (example).

This allows them to (theoretically) judge subtle differences between completions, at the cost of not being able to easily save and compare many different scores across prompts for the same test set. In addition, context length and memory limits can become an issue when comparing too long completions.

Some reward models such as SteerLM output absolute scores, which can be used to evaluate completions directly without the need for pairwise comparisions. These models can be easier to use for evaluation, but are also harder to collect data for, as absolute scores tend to be less stable than pairwise scores in human preferences.

More recently, models have been proposed that output both absolute and relative scores, such as HelpSteer2-Preference and ArmoRM.

> How do I use a Reward Model for Evaluation?
>
> Given a dataset of prompts, we can generate completions from a language model and ask a reward model to score them.
>
> For models that give absolute scores, the resulting scores can be averaged to get a reasonable summary score.
>
> However, in the more common case of relative scores, the average reward can be biased by outliers (a few very good or very bad completions) as different prompts may have inherently different reward scales (some prompts are way harder or easier than others).
>
> Instead, we can use
>
> - win rates: take a reference set of completions and calculate the percentage of completions from the model that are ranked higher than the reference completions. It is slightly more granular.
> - win probabilities: the mean probability of the completions being better than the reference completions, which can give a more fine-grained and smoothly changing signal.

> Pros and Cons of Reward Models
>
> Reward models are typically:
>
> - Very fast: Getting a score is as simple as running a forward pass of a relatively small model once (since we only get a score, and not long text, contrary to judge-LLMs)
> - Deterministic: The same scores will be reproduced through the same forward pass
> - Unlikely to suffer from positional bias: As most models take only one completion, they can not be influenced by the order. For pairwise models, positional bias is often also minimal, as long as the training data was balanced with respect to containing both first and second answers as being the best.
> - Require no prompt engineering: since the model will simply output a score from one or two completions depending on preference data it's been trained on.
>
> On the other hand they:

- Require specific fine-tuning: This can be a relatively costly step, and elthough they inherit many capabilities from a base model, they may still perform poorly on tasks that are out of the training distribution.
- Loose efficiency when used both in reinforcement learning and evaluation (or when using direct alignment algorithms on datasets that are similar to the training data of the reward model), as the language model may overfit to the reward model's preferences.

Going further

- A good place to find high performing models is the RewardBench Leaderboard.
- You can look at how reward models have been used in the Nemotron paper.
- For reward models that rate single prompts and completions, you can cache the scores of many reference models and easily see how a new model performs.
- Tracking of win rates or probabilities over training, e.g. as in this paper, can allow you to detect model degradation and select optimal checkpoints.

Constraining model outputs

In a number of cases, we might want the model to output a prediction which follows a very specific format to simplify evaluation.

USING A PROMPT

The easiest way to do this is to add a task prompt which contains very specific instructions as to how the model should answer (`Provide numerical answers in digits.`, `Use no abbreviation.`, etc).

It won't necessarily work all the time but should be good enough for high capability models. That's the approach we followed in the GAIA paper for example.

FEW SHOTS AND IN CONTEXT LEARNING

The next way to do so is to constrain the model through what is called "in context learning". By providing examples in the prompt (what is called `few-shot prompting`), the model is implicitly biased towards following the repeated prompt shape for the actual sample.

It's a method which was overall working quite well until end of 2023!

However, the widespread adoption of instruction-tuning methods and the addition of instruction data in later stages of model pre-training (continuous pre-training) has biased more recent models towards specific output formats (what is being called here *Training on the test task*, and what I would call *overfitting the prompt format*). Reasoning models are also not playing that well with few shot examples because of the reasoning trace.

It's also a method which can be limited for older models with smaller context sizes, as some few-shot examples can not fit into the context window.

STRUCTURED TEXT GENERATION

Structured text generation constrains the outputs to follow a given path, defined by a grammar or by regular expressions, for example. The `outlines` library implements this using finite state machines, which is very neat. (Other approaches exist, such as using interleaved generation for json generation, but the FSM one is my favorite).

To understand more about what happens when using structured generation, you can check the blog we wrote together: structured generation reduce prompt variance in evaluation, and make results and rankings more stable. You can also check the overall `outlines` blog for interesting implementations and observations linked to structured generation.

However, some recent research seems to show that structured generation can lower model performance on some tasks (like reasoning), by moving the prior too far away from the expected probability distribution.

> 📚 Going further
>
> - ⭐ Understanding how Finite State Machine when using structured generation, by Outlines. Super clear guide on how their method works!
> - The outlines method paper, a more academic explanation of the above
> - Interleaved generation, another method to constrain generations for some specific output formats

The forgotten children of evaluation

Statistical validity

When reporting evaluation results, it's critical to include confidence intervals alongside point estimates.

These confidence intervals from the raw scores can be obtained from standard deviations over the scores or bootstrapping - for automatic metrics, this is relatively trivial - for model judges, a recent paper suggested bias correction with estimators. For human based evaluations, you should report agreement.

You can also compute these with prompt variations, by asking the same questions in slightly different ways, or re-running on the same samples with different prompt formats.

Cost and efficiency

When designing and reporting evaluation results, we need to start collectively reporting results against model running costs! A reasoning model which requires 10 minutes of thinking and 10K tokens to answer 10 + 1 (because it decides to make an entire segue on binary vs decimal arithmetics) is considerably less efficient than a smol model answering 30 in a handful of tokens.

We suggest you report the following:

- Token consumption: Report the total number of output tokens used during evaluation. This is particularly important to estimate efficiency, and it will affect the cost of model as judge evaluations. Token counts directly impact monetary costs and help others estimate the computational requirements. Monetary cost can also be a good proxy for efficiency.
- Time: Document the inference time required by the model to complete the evaluation. This includes both the actual inference time and any overhead from API rate limits. This is particularly important for any time-sensitive applications (like some agentic tool use, as in GAIA2).

Last but not least, reporting the environmental footprint of the models you are running is becoming increasingly important with the overall state of resources available on earth. This includes carbon emissions from training and energy consumption at inference, and these will depend on the model size, hardware (if you know it) and the tokens generated. Some smaller or quantized models reach a very interesting performance to consumption ratio

Conclusion

Evaluation is both an art and a science. We've explored the landscape of LLM evaluation in 2025—from understanding why we evaluate models and the fundamental mechanics of tokenization and inference, to navigating the ever-evolving ecosystem of benchmarks, and finally to creating evaluations for your own use-cases.

Key things I hope you'll remember are:

Think critically about what you're measuring. Evaluations are proxies for capabilities, so a high score on a benchmark doesn't guarantee real-world performance. Different evaluation approaches (automatic metrics, human judges, or model judges) each come with their own biases, limitations, and tradeoffs.

Match your evaluation to your goal. Are you running ablations during training? Use fast, reliable benchmarks with strong signal even on small models. Comparing final models for selection? Focus on harder, uncontaminated datasets that test holistic capabilities. Building for a specific use case? Create custom evaluations that reflect your problems and data.

Reproducibility requires attention to detail. Small differences in prompts, tokenization, normalization, templates, or random seeds can swing scores by several points. When reporting results, be transparent about your methodology. When trying to reproduce results, expect that exact replication will be extremely challenging even if you attempt to control for every variable.

Prefer interpretable evaluation methods. When possible, functional testing and rule-based verifiers should be chosen over model judges. Evaluations that can be understood and debugged will provide clearer and more actionable insights… and the more interpretable your evaluation, the more you can improve your models!

Evaluation is never finished. As models improve, benchmarks saturate. As training data grows, contamination becomes more likely. As use cases evolve, new capabilities need measuring. Evaluation is an ongoing battle!

To conclude: The models we build are only as good as our ability to measure what matters. Thanks for reading!

Acknowledgments

Many thanks to all the people who contributed directly or indirectly to this document, notably Hynek Kydlicek, Loubna Ben Allal, Sander Land and Nathan Habib.

Citation

For attribution in academic contexts, please cite this work as

```
Clémentine Fourrier, Thibaud Frere, Guilherme Penedo, Thomas Wolf (2025). "The LLM Evaluation Guidebook".
```

BibTeX citation

```
@misc{fourrier2025_the_llm_evaluation_guidebook,
  title={The LLM Evaluation Guidebook},
  author={Clémentine Fourrier and Thibaud Frere and Guilherme Penedo and Thomas Wolf},
  year={2025},

}
```