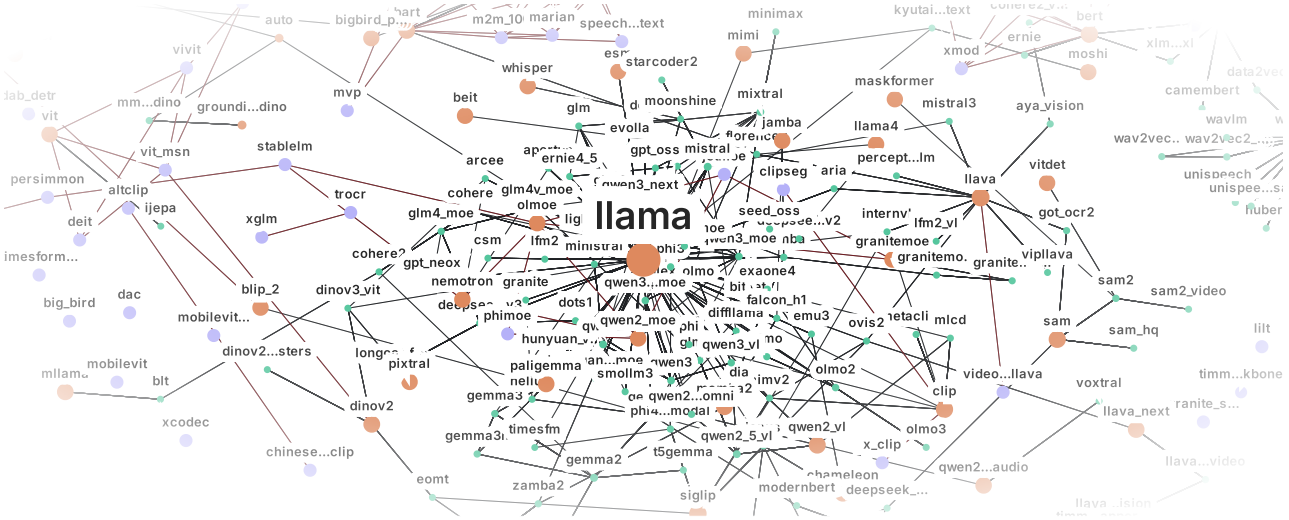
Maintain the unmaintainable: 1M python loc, 400+ models



A peek into software engineering for the transformers library

AUTHORS

Pablo Montalvo, Lysandre Debut, Pedro Cuenca, Yoni Gozlan

AFFILIATION

Hugging Face

PUBLISHED

October 6, 2025

Table of Contents

Table of Contents

Preface

One million lines of `Python` code. Through them, the `transformers` ↗ library supports more than 400 model architectures, from state-of-the-art LLMs and VLMs to specialized models for audio, video, and tables.

Built on `PyTorch`, it's a foundational tool for modern LLM usage, research, education, and tens of thousands of other open-source projects. Each AI model is added by the community, harmonized into a consistent interface, and tested daily on a CI to ensure reproducibility.

This scale presents a monumental engineering challenge.

How do you keep such a ship afloat, made of so many moving, unrelated parts, contributed to by a buzzing hivemind? Especially as the pace of ML research accelerates?

We receive constant feedback on everything from function signatures with hundreds of arguments to duplicated code and optimization concerns, and we listen to all of it, or try to. The library's usage keeps on growing, and we are a small team of maintainers and contributors, backed by hundreds of open-source community members. We continue to support all new models and expect to do so for the foreseeable future.

This post dissects the design philosophy that makes this possible. It's the result of an evolution from our older principles, detailed on our previous philosophy ↗ page, as well as its accompanying blog post from 2022 ↗. More recently (and we strongly recommend the read) we publish a blog post about recent upgrades to transformers ↗, focusing on what makes the library faster today. All of these developments are only made possible thanks to these principles.

We formalize and articulate the "tenets" that have been guiding our development, demonstrate how they are implemented in code, and show the measurable impact they have on the library's sustainability and growth.

For any OSS maintainer, power user, or contributor, this is the map to understanding, using, and building upon `transformers`, but not only: any project of comparable size will require you to make deep choices, not only on design and choice of abstraction, but on the very mindset of the software you are building. These tenets may or may not be applicable to your project, but they provide a glimpse on how we work that could be helpful or inspirational.

Conventions used throughout this post:

Tenets exemplified will have their summary available on hover.

External links ↗ to articles will help you solidify your knowledge.

Several interactive visualisations are available as you go - scroll, zoom, drag away to explore them.

> Breadcrumb boxes summarize what you just learned, connect it to the tenets, and point to what's coming Next. Think of them as narrative signposts to help you keep track.

We get started by enumerating the tenets. Then we look at concrete examples that show how they shape our decision-making. These examples are necessarily detailed, and sometimes complex, because they illustrate the challenges to maintain and grow a large codebase that

caters to multiple collectives, has millions of users, hundreds of contributors, and always strives for simplicity and consistency.

The core tenets of transformers

We summarize the foundations on which we've built everything, and write the "tenets" of the library. They behave like *software interfaces*, hence it is crucial that they are explicitly written down. However opinionated they are, they have evolved over time.

These principles were not decided in a vacuum. The library *evolved* towards them, and once they *emerged*, they were recognized as critical.

1 Source of Truth

We aim to be the source of truth for all model definitions ↗. This is not a tenet, but something that guides our decisions. Model implementations should be reliable, reproducible, and faithful to the original performances.

> *This overarching guideline ensures quality and reproducibility across all models in the library.*

2 One Model, One File

All inference and training core logic has to be visible, top-to-bottom, to maximize each model's hackability.

> *Every model should be understandable and hackable by reading a single file from top to bottom.*

3 Code is the Product

Optimize for reading, diff-ing, and tweaking, our users are power users. Variables can be explicit, full words, even several words, readability is primordial.

> *Code quality matters as much as functionality - optimize for human readers, not just computers.*

4 Standardize, Don’t Abstract

If it’s model behavior, keep it in the file; abstractions are only for generic infra.

> *Model-specific logic belongs in the model file, not hidden behind abstractions.*

5 DRY* (DO Repeat Yourself)

Copy when it helps users; keep successors in sync without centralizing behavior.

Evolution:

With the introduction and global adoption of modular transformers, we do not repeat any logic in the modular files, but end user files remain faithful to the original tenet.

> *Strategic duplication can improve readability and maintainability when done thoughtfully.*

6 Minimal User API

Config, model, pre-processing; from_pretrained, save_pretrained, push_to_hub. We want the least amount of codepaths. Reading should be obvious, configurations should be obvious.

> *Keep the public interface simple and predictable, users should know what to expect.*

7 Backwards Compatibility

Evolve by additive standardization, never break public APIs.

Any artifact that was once on the hub and worked with transformers should be usable indefinitely with the same interface. Further, public methods should not change to avoid breaking dependencies.

> *Once something is public, it stays public, evolution through addition, not breaking changes.*

8 Consistent Public Surface

Same argument names, same outputs, hidden states and attentions exposed, enforced by tests. This is a goal as well as a tenet.

> *All models should feel familiar - consistent interfaces reduce cognitive load.*

When a PR is merged, it is because the contribution is worthwhile, and because the `transformers` team finds the design of the contribution to be aligned with the tenets.

Does all the code in the library strictly follow these tenets? No. The library is a gigantic house with connected nooks, corridors, crannies everywhere, built by thousands of different workers. We *try* to make it so all the code added is compliant, because if we fail and merge it, we cannot change it lest we break backwards compatibility .

To see what constitutes adherence to the tenets, let's take the example of code repetition.

The following function, essential to the implementation of Rotary Positional Embeddings ↗ can be found in more than 70 `modeling_<file>.py` across `src/transformers/models/.` Why keep it? Because we want all the model logic to be contained in the modeling file . In order to do that, we do repeat ourselves .

```python
def rotate_half(x):
    """Rotates half the hidden dims of the input."""
    x1 = x[..., : x.shape[-1] // 2]
    x2 = x[..., x.shape[-1] // 2 :]
    return torch.cat((-x2, x1), dim=-1)
```

We want all models to have self-contained modeling code.

Each core functionality *must* be in the modeling code, every non-core functionality *can* be outside of it.

This comes as a great cost. Enter the `#Copied from...` mechanism: for a long time, these comments were indicating that some code was copied from another model, saving time both for the reviewers and for the CI. But the LOC count kept creeping up. Each new model copied over hundreds of lines that we considered largely boilerplate, yet, we could not remove them.

We need to separate both principles that were so far intertwined, repetition and hackabilty .

What's the solution to this?

> TL;DR: Read the code in one place, one model, one file. . Keep semantics local (Standardize, Don't Abstract). Allow strategic duplication for end users (DRY*). Keep the public surface minimal and stable (Minimal API, Backwards Compatibility, Consistent Surface).
>
> Next: how modular transformers honor these while removing boilerplate.

Modular transformers

Transformers is an opinionated library. The previous philosophy ↗ page, and the blog post ↗ were already pointing at the drawbacks mentioned just above, which have been iteratively addressed. `modular` transformers was introduced ↗ to allow a form of inheritance without breaking the one model, one file rule.

We amended the principle of DRY* by progressively removing all pieces of code that were "copied from" another file.

It works as follows. In order to contribute a model, `GLM` for instance, we define a `modular_` file that can inherit from *any function across all other modeling, configuration and processor files* already existing in the library. The modular file can use inheritance across models: and then, it is unravelled into a fully functional modeling file.

MODULAR_GLM.PY

```python
class GlmMLP(Phi3MLP):
    pass

class GlmAttention(LlamaAttention):
    def __init__(self, config, layer_idx=None):
        super().__init__(config, layer_idx)
        self.o_proj = nn.Linear(
            config.num_attention_heads * self.head_dim,
            config.hidden_size,
            bias=False
        )

class GlmForCausalLM(LlamaForCausalLM):
    pass
```

MODELING_GLM.PY (AUTO-EXPANDED)

```python
class GlmMLP(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config
        self.gate_up_proj = nn.Linear(
            config.hidden_size,
            2 * config.intermediate_size,
            bias=False
        )
        self.down_proj = nn.Linear(
            config.intermediate_size,
            config.hidden_size,
            bias=False
        )
```

Left: Clean modular definition with inheritance. Right: Auto-expanded version with all inherited functionality visible.

As you can see, we can define a new model as a *modular* combination of fragments taken from others.

You might think “well that’s just how inheritance works”. The crucial difference is that we do *visibly* what is essentially the *compiler*’s job: by unrolling the inheritances, we make visible all of the modeling code, keeping it all in one piece.

You can see below the difference between `GlmAttention` and `LlamaAttention`, with the latter having been copied with minimal changes.

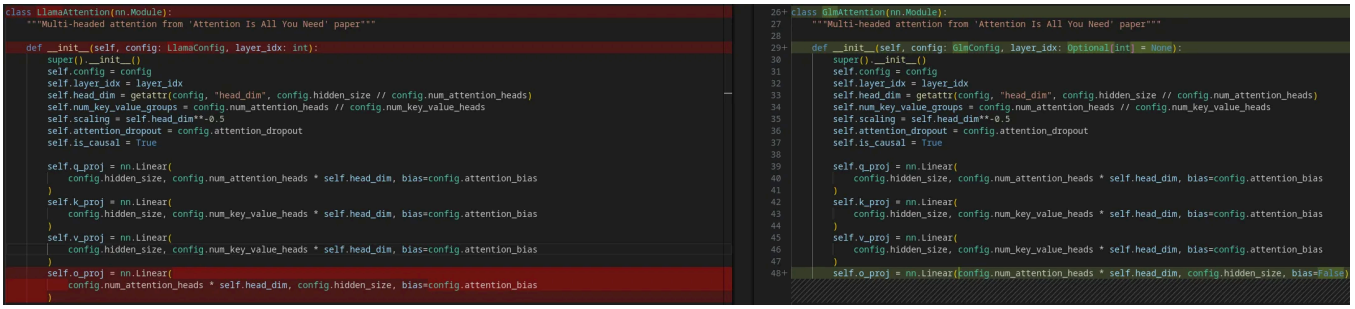


Figure 1: Comparison of attention implementations between Llama and GLM, showing code reuse with minimal modifications.

What is the consequence? When adding a model, we do not need to go over the entire modeling file. The modular (left side above) is enough.

When `AutoModel.from_pretrained(...)` is called, it is indeed the modeling (right side) that is run, and all the tests run on the modeling code.

More importantly, the auto-generated modeling file is what users *read* to understand the code, what they step through in their debuggers and what they hack for their needs.

What does that give us?

> TL;DR: A small `modular_*.py` declares reuse; the expanded modeling file stays visible and unique. Reviewers and contributors maintain the shard, not the repetition.
>
> Next: the measurable effect on effective LOC and maintenance cost.

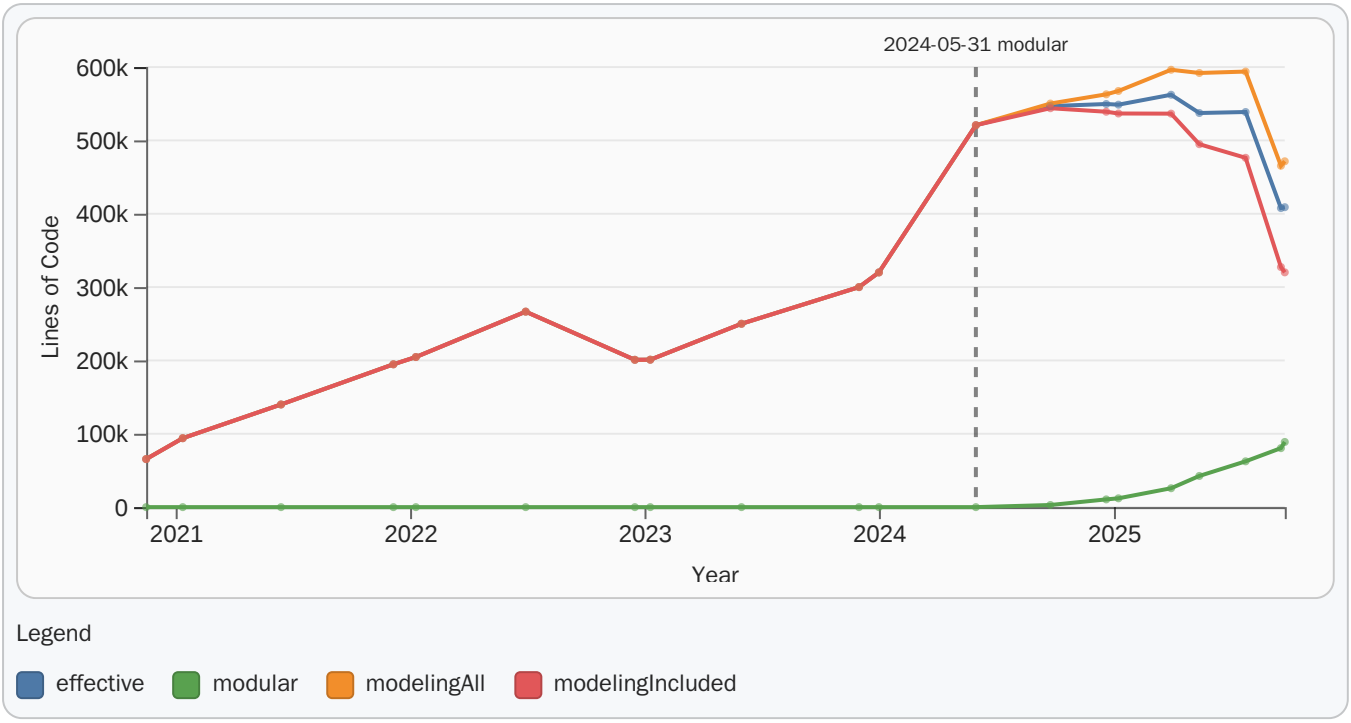
A maintainable control surface

The effect of modular can be measured in lines of code (LOC). If a model only has a modeling file, we add its LOC count. However, if a model has a `modular_*.py` and a corresponding automatically generated `modeling_*.py`, we only count the LOC under the modular file. The modeling code has no maintenance cost as it is strictly dependent on the modular file.

That gives an “effective LOC” curve: the **maintenance surface**.

Measured on git history, raw `modeling_*.py` grew at ~362 LOC/day before modular; counting only modular shards yields ~25 LOC/day after — about 15× lower. The effective curve (blue line below) represents the maintenance surface today: what maintainers actually read and review.

Less code to hand-maintain means fewer places to break. Naturally LOC is not a direct measure of complexity, but they correlate in review effort and change risk.



The blue line (effective) is the sum of the red + green, whereas the yellow would have been the progression without modular. We can see that the maintenance surface is essentially constant (in LOC) since the implementation of `modular`. If you zoom in, you'll notice there's a sharp drop near the end, it's essentially due to us removing support for Jax and TensorFlow ↗ library-wide.

But this was not the only effort that allowed us to reduce maintenance load.

We recently underwent a deep refactor of the attention implementation. You've likely heard about flash attention ↗ and its several variants.

The *attention computation* itself happens at a *lower* level of abstraction than the model itself.

However, we were adding specific torch operations for each backend (sdpa, the several flash-attention iterations, flex attention) but it isn't a minimal user api . Next section explains what we do.

> Evidence: effective (i.e., maintainable) LOC growth drops ~15× when counting shards instead of expanded modeling files. Less code to read, fewer places to break.

> Next: how the attention interface stays standard without hiding semantics.

External Attention classes

The solution for the “attention abstraction problem” was to move to a standard attention interface ↗ that allows the following:

The naive implementation of attention, called “eager”, is available by default. We use a `Callable` called `eager_attention_forward`, which can run as long as the user has PyTorch installed – which is a requirement any way.

Instead of using a class interface and a class hierarchy, we just moved to a function interface. When a more complex attention implementation is needed, we use other Callables, including much faster kernel bindings when available. The decision to use a different attention implementation is based on the model configuration file we download from the Hub, and it can also be overridden by the user.

This is a clear example that that we prefer an interface that is standard, but not abstract . To be completely precise, this is what the interface selection looks like in transformers code:

```
attention_interface: Callable = eager_attention_forward
if self.config._attn_implementation != "eager":
    attention_interface =
ALL_ATTENTION_FUNCTIONS[self.config._attn_implementation]
```

Having the attention interfaces functionalized allows to do dynamic switching of attentions as well, increasing their hackability . Another strength of the new attention interface is the possibility to enforce specific kwargs, which are needed by kernel providers and other dependencies.

Backend integrations sometimes require specific kwargs.

We know that kwargs are often a necessary evil that plagues tools with widespread compatibility; and it is something we have aimed to reduce, and continue to reduce in order to improve readability - with them, the current system is a minimal user api .

We reduce that surface and document expectations; where flexibility is necessary, we plan to use `typing.Annotated` to convey shapes and invariants without constraining integrations. Such an implementation could look like this in the future:

```python
from typing import Annotated

MyModelOutputAnnotated = Annotated[MyModelOutput, "shape: (B, C, H, W)"]
```

> Attention semantics remain in `eager_attention_forward`; faster backends are opt-in via config. We inform via types/annotations rather than enforce rigid kwargs, preserving integrations.
>
> Next: parallel partitioning is declared as a plan, not through model surgery.

Configurable Tensor Parallelism

If you're not familiar with the different flavours of parallelism, I recommend checking out this blog post ↗ first, and of course a full dive into the ultra-scale playbook ↗ is always recommended.

The essential part is that, as the documentation states ↗, when tensors get too large to fit on a single GPU, they are sliced along a particular dimension and every slice is sent to a different GPU.

Why does it matter?

Because we want to avoid code modifications that are unrelated to the model.

We choose to place the level of abstraction higher than the device placement: a matrix multiplication - a `nn.Linear` layer - should be always expressed in the same way, regardless of how it is placed.

Hence, we want to touch the modeling code as little as possible , and only modify it when *architectural changes* are involved — not depending on the way you run it. For tensor parallelism, we simply specify a `tp_plan`:

```python
# In the model's config (example: ERNIE 4.5-style decoder blocks)
base_model_tp_plan = {
    "layers.*.self_attn.q_proj": "colwise",
    "layers.*.self_attn.k_proj": "colwise",
    "layers.*.self_attn.v_proj": "colwise",
    "layers.*.self_attn.o_proj": "rowwise",
    "layers.*.mlp.gate_proj": "colwise",
    "layers.*.mlp.up_proj":   "colwise",
    "layers.*.mlp.down_proj": "rowwise",
}

# Runtime
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer

model_id = "your/model-or-local-checkpoint"
model = AutoModelForCausalLM.from_pretrained( # <-- will automatically map to the plan defined above
    model_id,
    dtype=torch.bfloat16,
)
tok = AutoTokenizer.from_pretrained(model_id)
inputs = tok("Hello", return_tensors="pt").to(model.device)
out = model(**inputs)
```

The plan is written once, saved as part of the config and passed to `.from_pretrained()`. It maps module name patterns to partitioning strategies. Strategies are resolved by the internal `ParallelInterface`, which wires to sharding implementations `ColwiseParallel`, `RowwiseParallel`, packed variants, and so on.

The alternative would be to modify classes depending on supported types of parallelism.

The `tp_plan` solution allows users to run the same model on a single GPU, or distribute it using multiple processes per node, e.g. 4 GPUs:

`torchrun --nproc-per-node 4 demo.py`

Semantics stay in the model (a Linear stays a Linear), parallelization is orthogonal and declared via strings: “colwise” splits columns of weights/bias across ranks; “rowwise” splits rows; packed variants shard fused weights; The mapping keys accept glob patterns like `layers.*.mlp.down_proj` to target repeated submodules.

> Parallelization is specified in the configuration (`tp_plan`), not through edits to `Linear`s. Glob patterns target repeated blocks; modeling semantics stay intact.

> Next: per-layer attention/caching schedules declared in config, not hardcoded.

Layers, attentions and caches

Following the same logic, the *nature* of attention and caching per layer of a model should not be hardcoded. We should be able to specify in a configuration-based fashion how each layer is implemented. Thus we define a mapping that can be then

```
ALLOWED_LAYER_TYPES = (
    "full_attention",
    "sliding_attention",
    "chunked_attention",
    "linear_attention",
    ...
)
```

and the configuration can be *explicit* about which attention type is in which layer. See, for example, gpt-oss ↗, which alternates sliding and full attention:

```
  "layer_types": [
    "sliding_attention",
    "full_attention",
    ...,
    "sliding_attention",
    "full_attention"
  ],
```

This is minimal to implement on the user side, and allows to keep the modeling code untouched. It is also easy to tweak.

> Allowed layer types are explicit; schedules (e.g., sliding/full alternation) live in config. This keeps the file readable and easy to tweak.
>
> Next: speedups come from kernels that don't change semantics.

Community Kernels

The same principle extends to normalization, activation, and other code paths. The model defines semantics; a kernel defines how to execute them faster. We annotate the module to

borrow a community-provided forward, keeping a consistent public surface

```python
@use_kernel_forward_from_hub("RMSNorm")
class GlmRMSNorm(nn.Module):
    ...
```

This also opens another contribution path: GPU specialists can contribute optimized kernels to the Kernels Hub ↗, and have them immediately available to use in `transformers` and other libraries. You can check the kernel community blog post ↗ to learn more about it!

Even more resources have been added, like the formidable kernel builder ↗ with its connected resources to help you build kernels with it ↗ and with nix ↗.

> Models define semantics; kernels define how to run them faster. Use decorations to borrow community forwards while keeping a consistent public surface.
>
> Next: what modularity looks like across the repo.

A Modular State

With `modular` transformers, we have a form of inheritance in our codebase. Some models become standards, and model contributors are given the opportunity to *define standards*. Pushing the boundaries of scientific knowledge can translate into the boundaries of engineering if this effort is made, and we're striving for it. It's hard to conceptualize very large libraries and how their components interact with each other, regardless of your cognitive abilities for abstractions. So I want to take a look at the current state of modularity across the repository. How many models are defined using components of others?

To get this graph, I use the heuristic of modular inheritance.

1. Does this model have a `modular` file?
2. In this `modular` file, what models, configurations and processings are imported?
3. Recurse through the model list that way.

So what do we see?

(Graph reading guide: nodes are models; edges are modular imports).

Check out the full viewer here ↗ (tab “dependency graph”, hit “build graph”) for better manipulation and exploration.

[Interactive content - view online]

Let’s walk through some sections of this graph together. First, Llama is a basis and an influence for many models, and it is very visible.

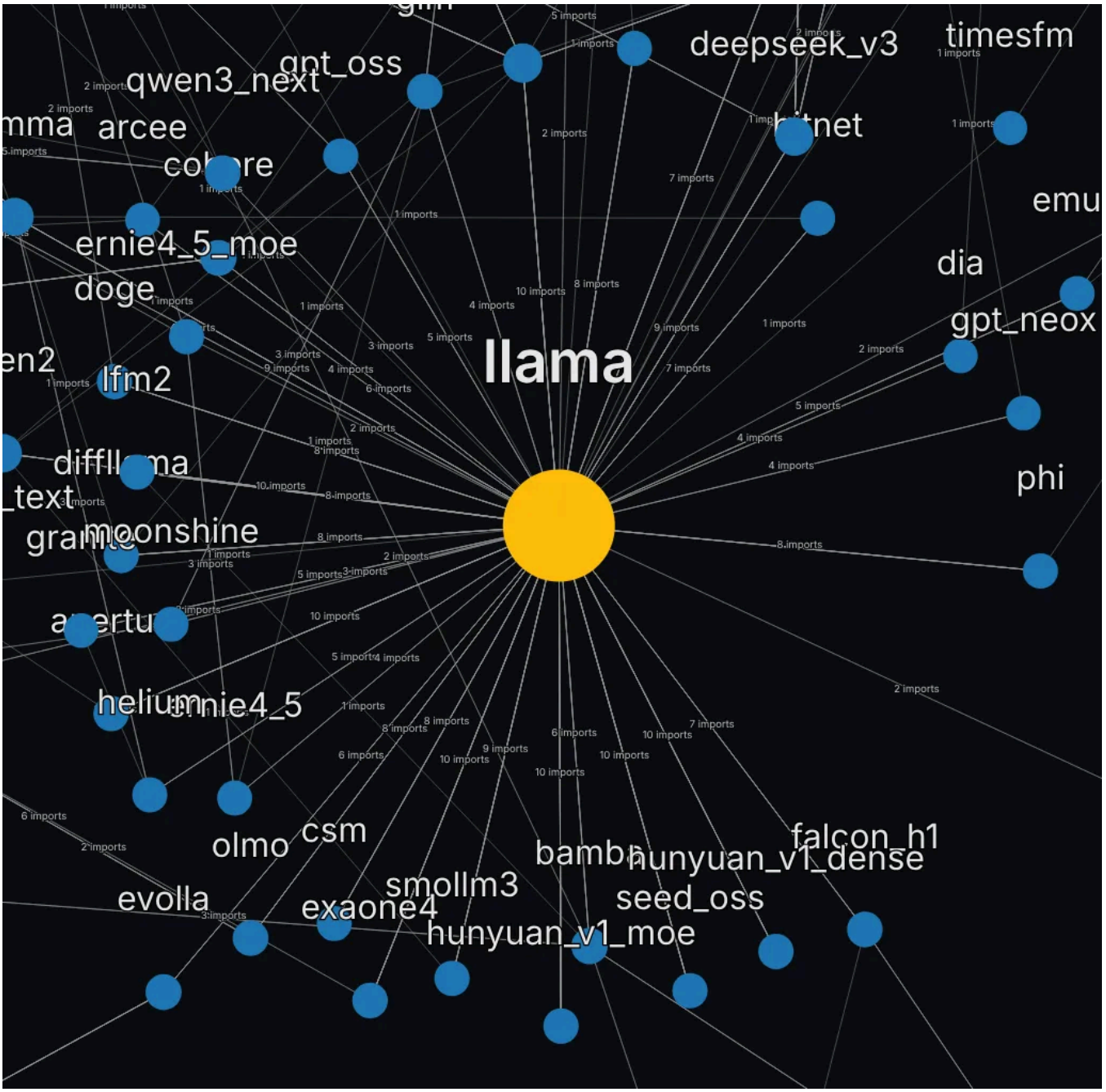


Figure 2: Llama as a central model influencing many other models in the dependency graph.

The models linked sometimes pull components from other models than `llama` of course. Radically different architectures such as mamba have spawned their own dependency subgraph.

Audio models form sparser archipelagos, see for instance wav2vec2 which is a significant basis for a dozen of them.

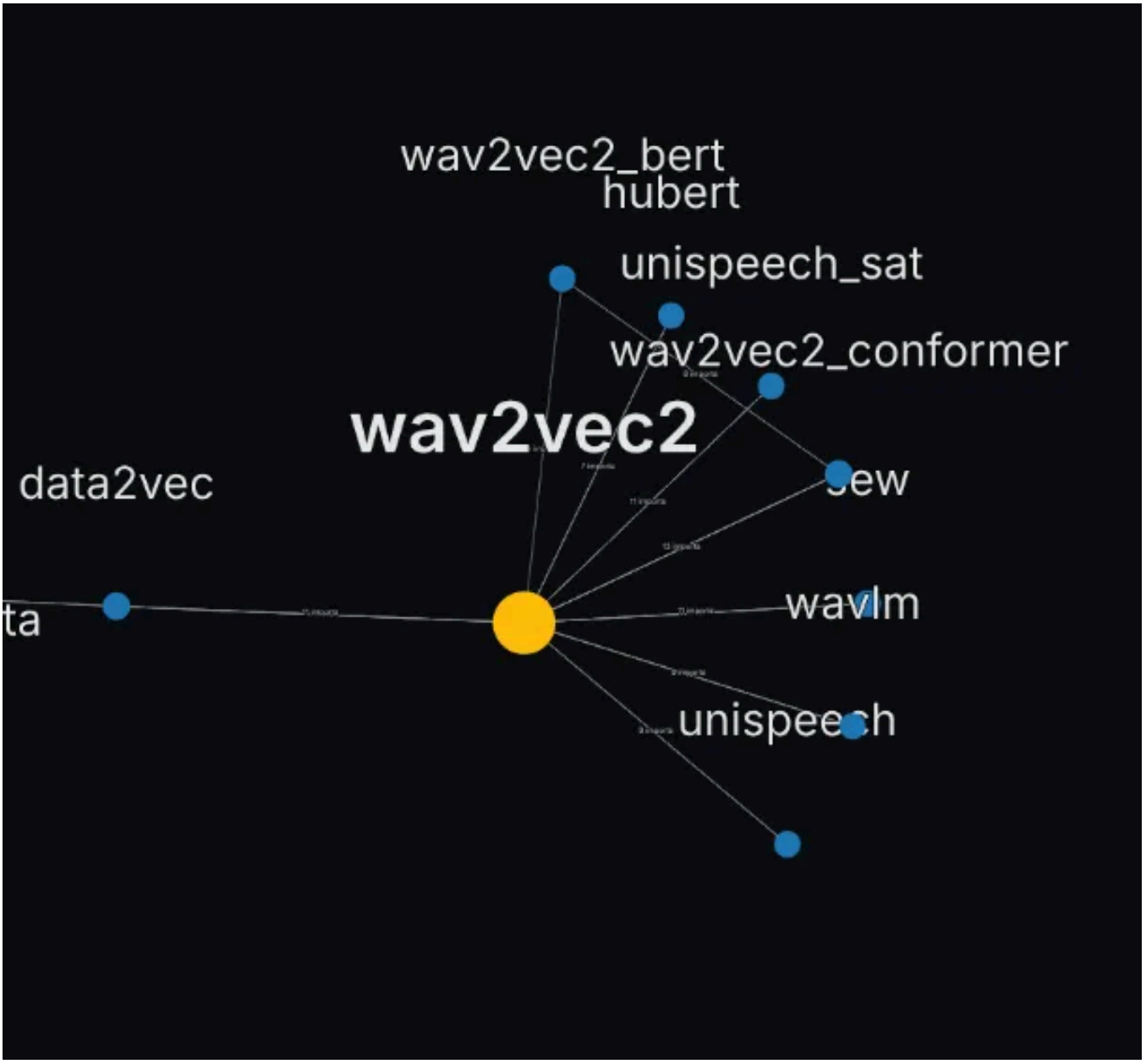


Figure 3: Cluster of audio architectures based on wav2vec2, forming a specialized archipelago.

In the case of VLMs which have massively grown in popularity since 2024, there's far too many vision-based architectures that are not yet defined as modulars of other existing archs. In other words, there is no strong reference point in terms of software for vision models.

As you can see, there is a small `DETR` island:

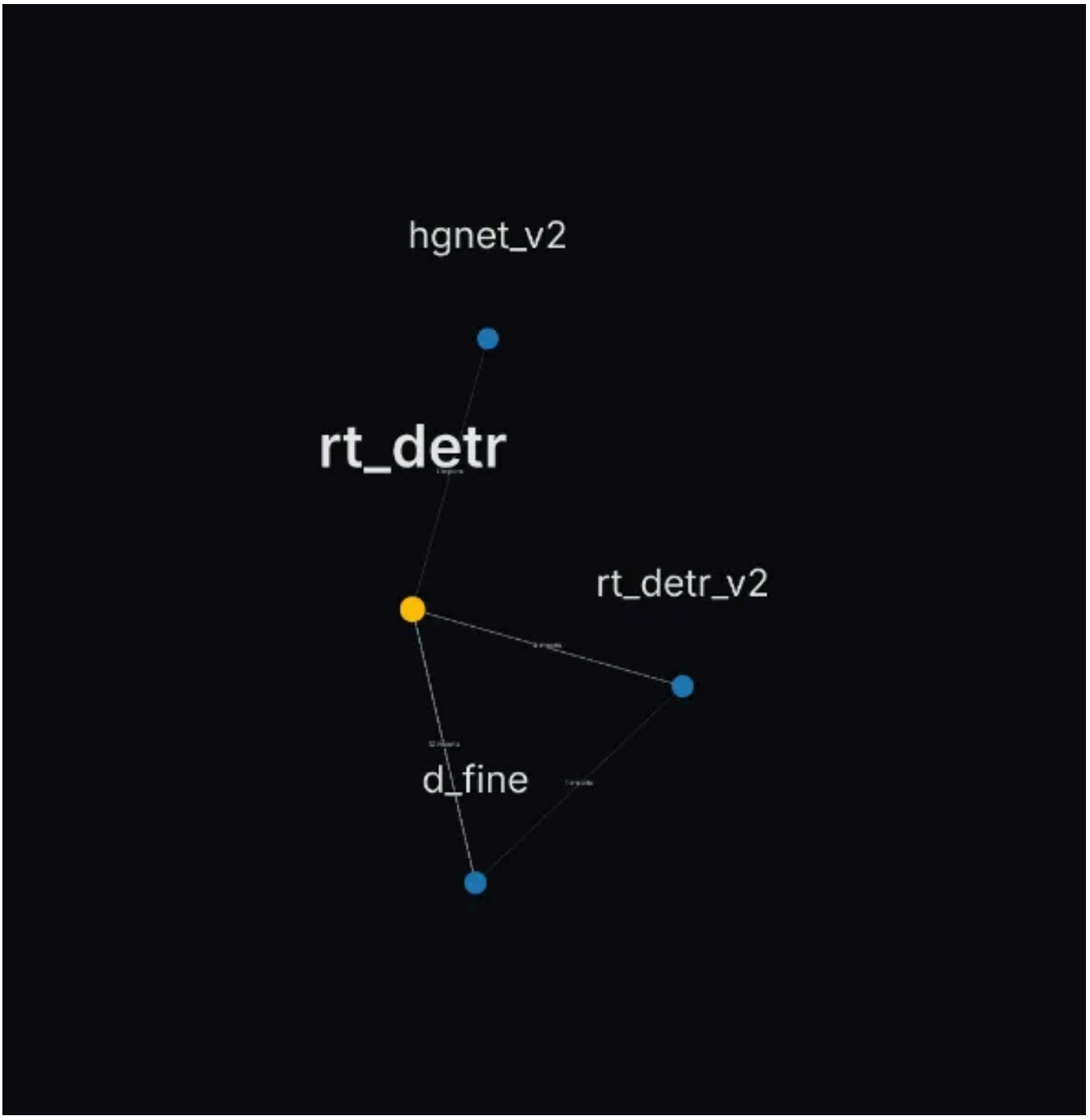


Figure 4: Small DETR archipelago for vision models, less centralized than Llama for text.

There is also a little llava pocket, and so on, but it's not comparable to the centrality observed for llama.

Another problem is, this visualization only shows `modular` models. Several models still do NOT have a modular file. If we zoom out significantly, we can see them, the red nodes are models that do not have a modular file yet.

Figure 5: Overview showing red nodes (models without modular files) to be modularized.

Hence the next question, and how do we identify modularisable models?

> Llama-lineage is a hub; several VLMs remain islands — engineering opportunity for shared parents. Next: timeline + similarity signals to spot modularisable candidates.

Many models, but not enough yet, are alike

Next, I looked into Jaccard similarity, which we use to measure set differences. I know that code is more than a set of characters stringed together. I also used code embedding models to check out code similarities, and it yielded better results, for the needs of this blog post I will stick to Jaccard index.

It is interesting, for that, to look at *when* we deployed this modular logic and what was its rippling effect on the library. You can check the larger space ↗ to play around, but the gist is: adding modular allowed to connect more and more models to solid reference points. We have a lot of gaps to fill in still.

Zoom out below - it's full of models. You can click on a node to see its connections better, or use the text box to search for a model. You can use the full viewer ↗ (tab "timeline", hit "build timeline") for better exploration.

[Interactive content - view online]

Let's look at a few highly connected models. Let's start by the foundational work of Llava ↗.

Figure 6: LLaVA and its variants in the timeline, with llava_video as a candidate for modularization.

You see that `llava_video` is a red node, connected by a red edge to `llava`: it's a candidate, something that we can *likely* remodularize, not touching the actual model but being much more readable with DRY* .

The same can be identified with the classical encoders family, centered on `BERT`:

Here `roberta`, `xlm_roberta`, `ernie` are `modular`s of BERT, while models like `mobilebert` are likely candidates.

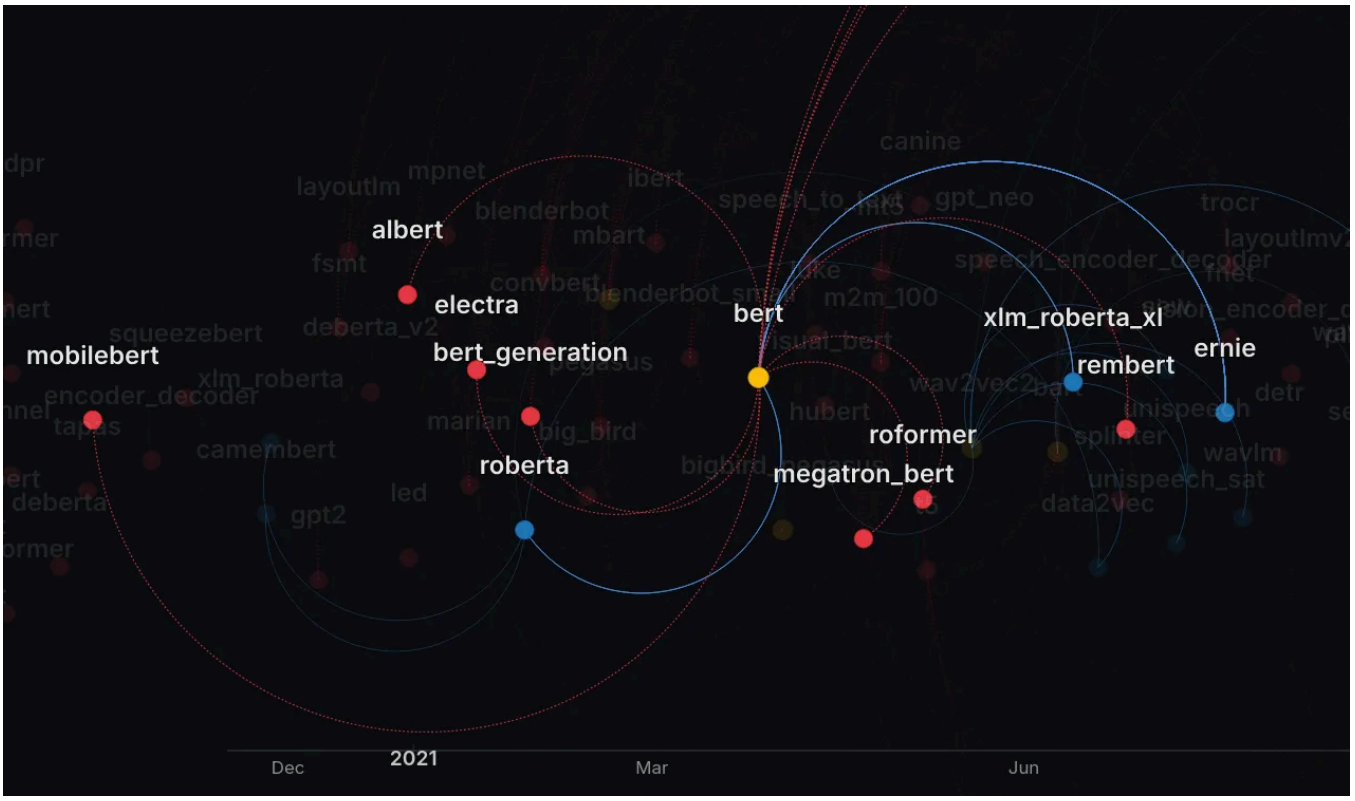


Figure 7: Family of classical encoders centered on BERT, with several models already modularized.

> Similarity metrics (Jaccard index or embeddings) surfaces likely parents; the timeline shows consolidation after modular landed. Red nodes/edges = candidates (e.g., `llava_video` → `llava`) for refactors that preserve behavior.
>
> Next: concrete VLM choices that avoid leaky abstractions.

VLM improvements, avoiding abstraction

We don't yet have a cookbook for common VLM patterns (image token scatter, multi-tower encoders, cross-attention bridges). This is one of the main improvement points where we can work.

For instance, we thought of abstracting away the mixing of `inputs_embeds`, the tensor fed into an LLM decoder in 95% of the existing VLMs. It would have looked like something like

```python
class InputsEmbeddingMixerMixin(nn.Module):
    #
```

But this is not an abstraction . Embedding mixin is part of the model, removing it would break it. A user opening `modeling_qwen2.5_vl` ↗ (check out the Qwen2.5VL collection ↗) should not have to go to another file to understand how it works.

What is the current state of these “abstractions” across the codebase? You will see all the imports around a modeling file, here Gemma3n ↗.

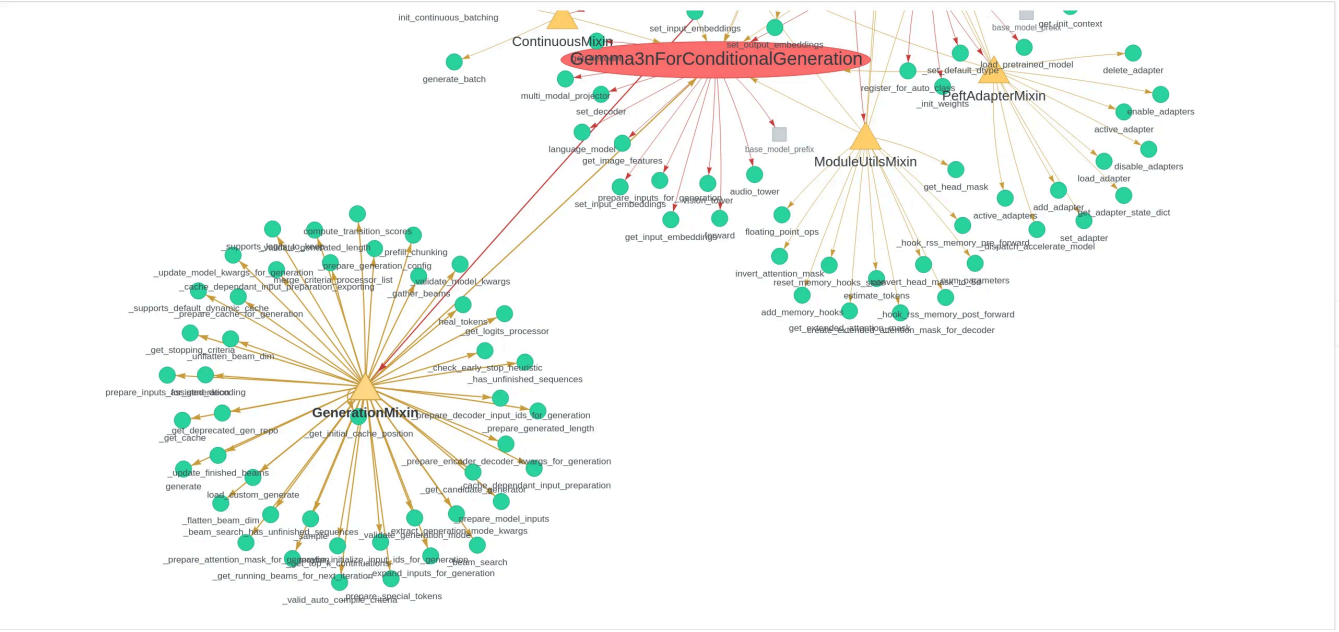


Figure 8: Gemma3n import graph showing dependency complexity, with GenerationMixin very central.

As you can see, the `GenerationMixin` node is already very heavy. It encompasses all of the utilities around `.generate`, it is second only to `nn.Module`. That means every decision we make to abstract something else has to be extremely careful.

The following Pull request to standardize placeholder masking ↗ is a good example of what kind of changes are acceptable. In a VLM, we always need to insert embeddings from various encoders at various positions, so we can have a function to do it. For Qwen2 VL, for instance, it will look like this:

```
    def get_placeholder_mask(
        self,
        input_ids: torch.LongTensor,
        inputs_embeds: torch.FloatTensor,
        image_features: torch.FloatTensor = None,
        video_features: torch.FloatTensor = None,
    ):
        """
        Obtains multimodal placeholdr mask from `input_ids` or `inputs_embeds`, and checks that the placeholder token count is
        equal to the length of multimodal features. If the lengths are different, an error is raised.
        """
        if input_ids is None:
            special_image_mask = inputs_embeds == self.get_input_embeddings()(
                torch.tensor(self.config.image_token_id, dtype=torch.long, device=inputs_embeds.device)
            )
            special_image_mask = special_image_mask.all(-1)
            special_video_mask = inputs_embeds == self.get_input_embeddings()(
                torch.tensor(self.config.video_token_id, dtype=torch.long, device=inputs_embeds.device)
            )
            special_video_mask = special_video_mask.all(-1)
        else:
            special_image_mask = input_ids == self.config.image_token_id
            special_video_mask = input_ids == self.config.video_token_id

        n_image_tokens = special_image_mask.sum()
        special_image_mask = special_image_mask.unsqueeze(-1).expand_as(inputs_embeds).to(inputs_embeds.device)
        if image_features is not None and inputs_embeds[special_image_mask].numel() != image_features.numel():
            raise ValueError(
                f"Image features and image tokens do not match: tokens: {n_image_tokens}, features {image_features.shape[0]}"
            )

        n_video_tokens = special_video_mask.sum()
        special_video_mask = special_video_mask.unsqueeze(-1).expand_as(inputs_embeds).to(inputs_embeds.device)
        if video_features is not None and inputs_embeds[special_video_mask].numel() != video_features.numel():
            raise ValueError(
```

```
                f"Videos features and video tokens do not match: tokens:
{n_video_tokens}, features {video_features.shape[0]}"
            )

        return special_image_mask, special_video_mask
```

But this is *within* the modeling file, not in the `PreTrainedModel` base class. It does not move away from it, because it'd break the One model, one file tenet.

What do we conclude? Going forward, we should aim for VLMs to have a form of centrality similar to that of `Llama` for text-only models. This centrality should not be achieved at the cost of abstracting and hiding away crucial inner workings of said models.

> Keep VLM embedding mix in the modeling file (semantics), standardize safe helpers (e.g., placeholder masking), don't migrate behavior to `PreTrainedModel`. Next: pipeline-level wins that came from PyTorch-first choices (fast processors).

On image processing and processors

Deciding to become a `torch`-first library meant relieving a tremendous amount of support for `jax` and `TensorFlow`, and it also meant that we could be more lenient about the amount of torch-dependent utilities that we were able to accept. One of these is the *fast processing* of images. Where inputs were once minimally assumed to be ndarrays, enforcing native `torch` and `torchvision` inputs allowed us to massively improve processing speed for each model.

The gains in performance are immense, up to 20x speedup for most models when using compiled torchvision ops. Furthermore, let us run the whole pipeline solely on GPU.

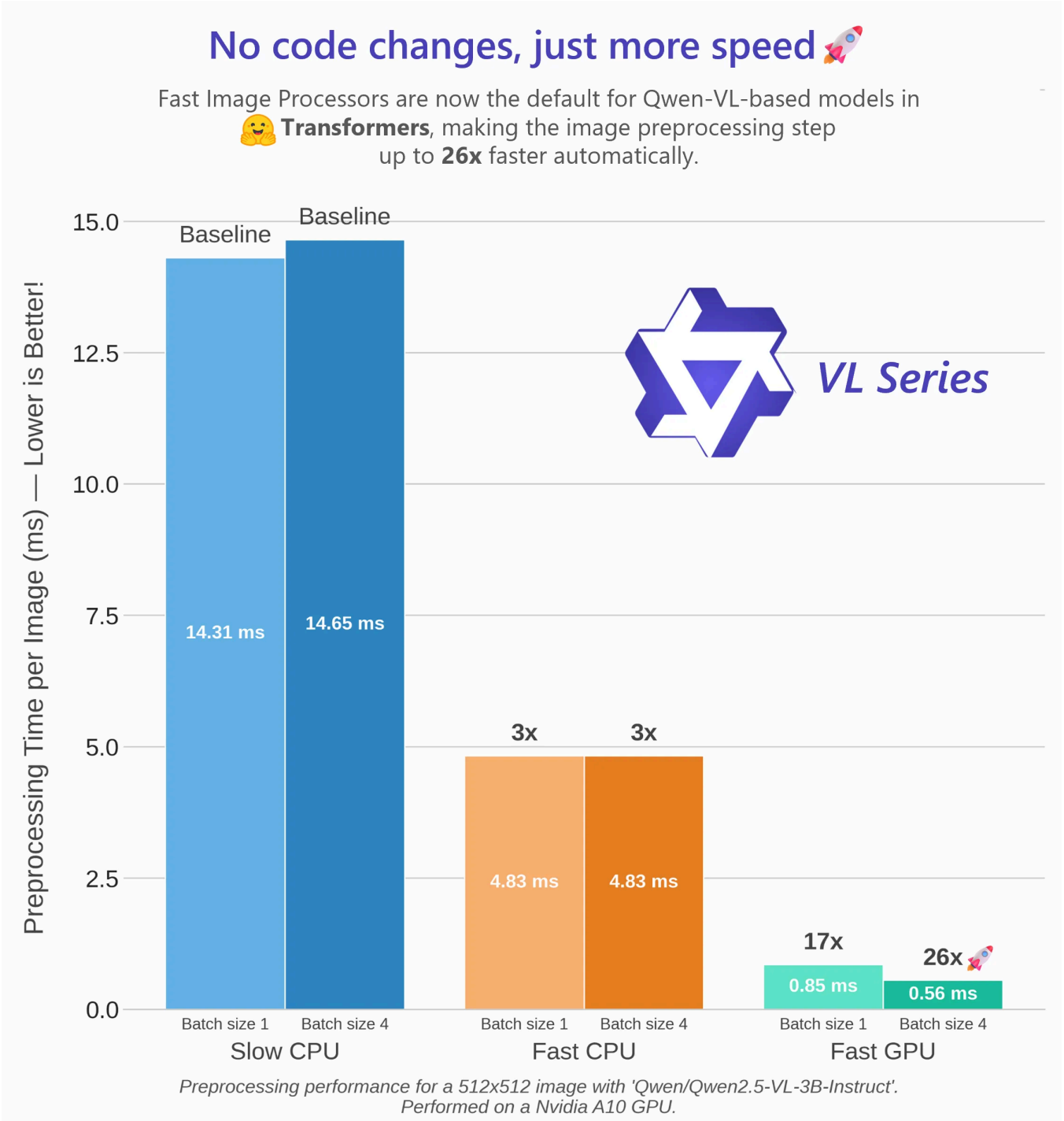


Figure 9: Performance gains of fast image processors, up to 20x acceleration with compiled torchvision.

PyTorch-first lets processors assume torch/torchvision and run the whole pipeline on GPU; big per-model speedups.

Next: how this lowers friction for contributors and downstream users.

Reduce barrier to entry/contribution

This is an overall objective: there's no `transformers` without its community.

Having a framework means forcing users into it. It restrains flexibility and creativity, which are the fertile soil for new ideas to grow.

Among the most valuable contributions to `transformers` is of course the addition of new models. Recently, OpenAI added GPT-OSS ↗, which prompted the addition of many new features to the library in order to support their model ↗.

A second one is the ability to fine-tune and pipeline these models into many other softwares. Check here on the hub how many finetunes are registered for gpt-oss 120b ↗, despite its size!

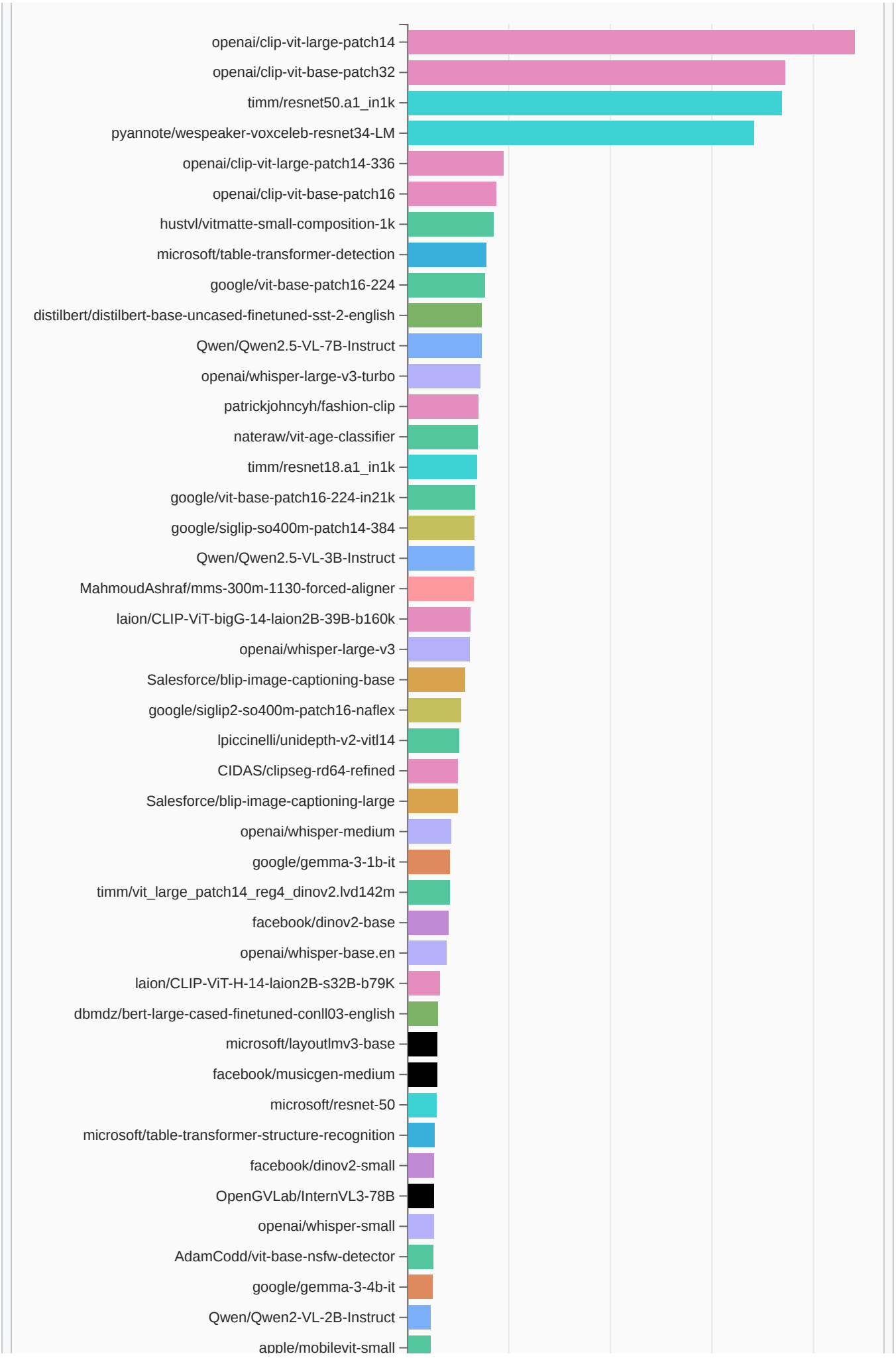
> The shape of a contribution: add a model (or variant) with a small modular shard; the community and serving stacks pick it up immediately. Popularity trends (encoders/embeddings) guide where we invest.
>
> Next: power tools enabled by a consistent API.

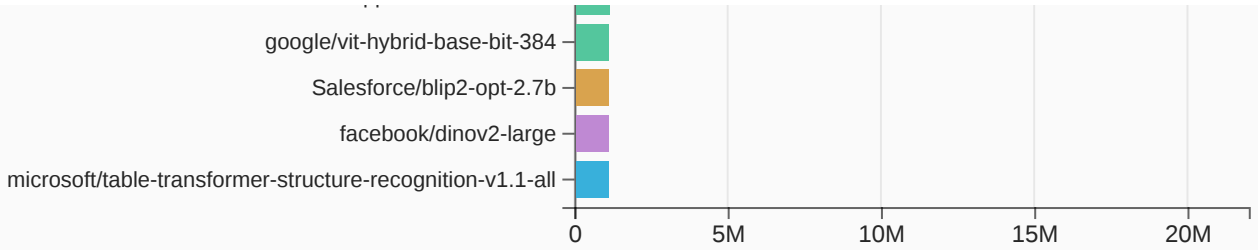
Models popularity

Talking about dependencies, we can take a look at the number of downloads as a measure of popularity. One thing we see is the prominence of encoders, despite the apparent prevalence of decoder LLMs. The reason is that encoders are used to generate embeddings, which have multiple downstream uses. Just check out EmbeddingGemma ↗ for a modern recap. Hence, it is vital to keep the encoders portion of the library viable, usable, fine-tunable.

Metric

Sort by Downloads


openai/clip-vit-large-patch14
openai/clip-vit-base-patch32
timm/resnet50.a1_in1k
pyannote/wespeaker-voxceleb-resnet34-LM
openai/clip-vit-large-patch14-336
openai/clip-vit-base-patch16
hustvl/vitmatte-small-composition-1k
microsoft/table-transformer-detection
google/vit-base-patch16-224
distilbert/distilbert-base-uncased-finetuned-sst-2-english
Qwen/Qwen2.5-VL-7B-Instruct
openai/whisper-large-v3-turbo
patrickjohncyh/fashion-clip
nateraw/vit-age-classifier
timm/resnet18.a1_in1k
google/vit-base-patch16-224-in21k
google/siglip-so400m-patch14-384
Qwen/Qwen2.5-VL-3B-Instruct
MahmoudAshraf/mms-300m-1130-forced-aligner
laion/CLIP-ViT-bigG-14-laion2B-39B-b160k
openai/whisper-large-v3
Salesforce/blip-image-captioning-base
google/siglip2-so400m-patch16-naflex
lpiccinelli/unidepth-v2-vitl14
CIDAS/clipseg-rd64-refined
Salesforce/blip-image-captioning-large
openai/whisper-medium
google/gemma-3-1b-it
timm/vit_large_patch14_reg4_dinov2.lvd142m
facebook/dinov2-base
openai/whisper-base.en
laion/CLIP-ViT-H-14-laion2B-s32B-b79K
dbmdz/bert-large-cased-finetuned-conll03-english
microsoft/layoutlmv3-base
facebook/musicgen-medium
microsoft/resnet-50
microsoft/table-transformer-structure-recognition
facebook/dinov2-small
OpenGVLab/InternVL3-78B
openai/whisper-small
AdamCodd/vit-base-nsfw-detector
google/gemma-3-4b-it
Qwen/Qwen2-VL-2B-Instruct
apple/mobilevit-small



As the codebase grows, we need to maintain it in coordination with our friend Sentence Transformers codebase ↗. Retrieval use-cases, smart databases, FAISS-based indexing rely on it, and thus indirectly on transformers.

In that regard, we DO want to be a modular toolbox, being minimal enough and well documented enough so any ML/AI developer can use `transformers` without having to think about it. We aim to reduce the cognitive load brought about by model development, not increase it.

So, how do these design choices, these “tenets” influence development of models and overall usage of transformers?

> Encoders remain critical for embeddings and retrieval; maintaining them well benefits the broader ecosystem (e.g., Sentence Transformers, FAISS).
>
> Next: dev tools that leverage unified attention APIs and PyTorch-only internals.

A surgical toolbox for model development

Transformers provides many tools that can help you add a new architecture, understand the inner workings of a model, as well as the library itself.

Attention visualisation

All models have the same API for attention computation, thanks to the externalisation of attention classes.

This uniformity allows us to build cool tools to visualize the inner workings of the attention mechanism.

One particular piece of machinery is the `attention mask`. Here you see the famous bidirectional attention pattern for the whole prefix (text + image) in PaliGemma and all Gemma2+ models, contrasting with the usual “causal-only” models.

attention-mask-visualizer

```
ATTN MASK — GPT-2 (causal)
Tokens: [The, cat, sat, on, the, mat]
Legend: x = can attend, . = masked (future)

         The cat sat on  the mat
The       x
cat       x   x
sat       x   x   x
on        x   x   x   x
the       x   x   x   x   x
mat       x   x   x   x   x   x


ATTN MASK — PaliGemma-style (bidirectional prefix + causal suffix)
Prefix:  [<i0> <i1> <i2> <i3> <i4> What is this]
Suffix:  [A great duck]
Legend: ✓ = can attend, ✗ = cannot

           <i0><i1><i2><i3><i4> What  is  this  |   A   great  duck
<i0>        ✓    ✓    ✓    ✓    ✓    ✓     ✓     ✓        ✗     ✗     ✗
<i1>        ✓    ✓    ✓    ✓    ✓    ✓     ✓     ✓        ✗     ✗     ✗
<i2>        ✓    ✓    ✓    ✓    ✓    ✓     ✓     ✓        ✗     ✗     ✗
<i3>        ✓    ✓    ✓    ✓    ✓    ✓     ✓     ✓        ✗     ✗     ✗
<i4>        ✓    ✓    ✓    ✓    ✓    ✓     ✓     ✓        ✗     ✗     ✗
What        ✓    ✓    ✓    ✓    ✓    ✓     ✓     ✓        ✗     ✗     ✗
is          ✓    ✓    ✓    ✓    ✓    ✓     ✓     ✓        ✗     ✗     ✗
this        ✓    ✓    ✓    ✓    ✓    ✓     ✓     ✓        ✗     ✗     ✗
-------------------------------------------------------------------------
A           ✓    ✓    ✓    ✓    ✓    ✓     ✓     ✓        ✓     ✗     ✗
great       ✓    ✓    ✓    ✓    ✓    ✓     ✓     ✓        ✓     ✓     ✗
duck        ✓    ✓    ✓    ✓    ✓    ✓     ✓     ✓        ✓     ✓     ✓
```

> Uniform attention APIs enable cross-model diagnostics (e.g., PaliGemma prefix bidirectionality vs causal).
>
> Next: whole-model tracing for ports and regressions.

Logging entire model activations

Because everything is PyTorch, we can easily debug any model ↗ when we want to add it to transformers. We now have a power-user tool for porting or adding models, that wraps a forward pass, intercepts every submodule call, and logs shapes, dtypes, and sample statistics of inputs/outputs to nested JSON.

It just works with PyTorch models and is especially useful when aligning outputs with a reference implementation, to match our Source of Truth guideline .

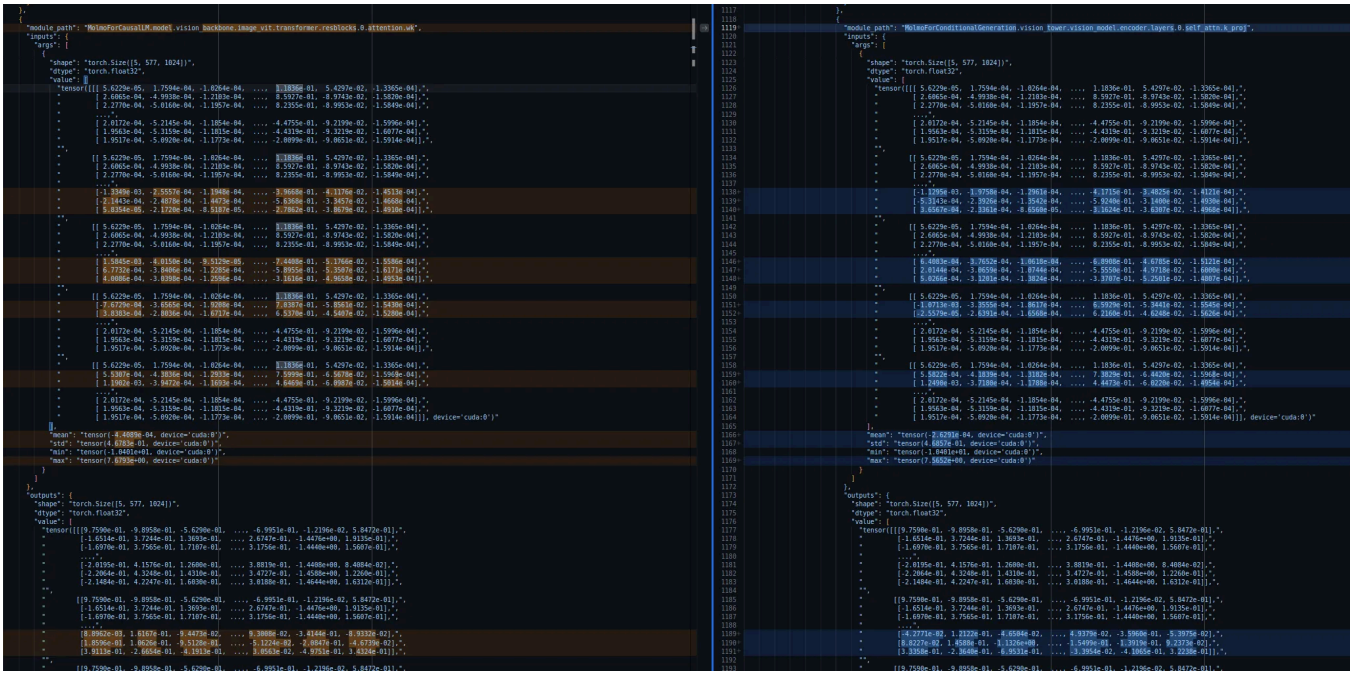


Figure 10: Model debugger interface intercepting calls and logging statistics in nested JSON.

> Forward interception and nested JSON logging align ports to reference implementations, reinforcing “Source of Truth.”
>
> Next: CUDA warmup reduces load-time without touching modeling semantics.

Cooking faster CUDA warmups

Having a clean *external* API allows us to work on the true inner workings of transformers . One of a few recent additions is the *CUDA warmup* via `caching_allocator_warmup`, which dramatically improves loading times by pre-allocating GPU memory to avoid malloc bottlenecks during model loading. It can achieve a 7x speedup factor for an 8B model, or 6x for a 32B one, as you can check in the PR ↗!

Mem allocation patterns during model loading

Start Animation Reset

0.00s

Layers loaded: 0/10

Individual Allocations:
Each model layer triggers a separate cudaMalloc() call, creating memory fragmentation and allocation overhead.

📦 **Grey "malloc"** = Memory allocation overhead
✅ **Green "data"** = Actual layer data loading

0.00s

Layers loaded: 0/10

Pre-allocated Memory Pool

Pre-allocated Pool:
The warmup function calculates total memory needed and makes ONE large allocation. Subsequent layers load directly into this pool, eliminating malloc overhead.

- **Container** = Single large malloc (warmup)
- **Progress bar** = Layer data loading (no malloc needed)

It's hard to overstate how much of a lifesaver that is when you're trying to load a model as fast as possible, as it's the narrowest bottleneck for your iteration speed.

> Pre-allocating GPU memory removes malloc spikes (e.g., 7× for 8B, 6× for 32B in the referenced PR).
>
> Next: consistent interfaces allow transformers-serve.

Transformers-serve and continuous batching

Having all these models readily available and sharing the same interface allows us to implement transformers-serve, a CLI tool to expose models through a standard OpenAI http API.

```
transformers serve

curl -X POST http://localhost:8000/v1/chat/completions \
-H "Content-Type: application/json" \
-d '{"messages": [{"role": "system", "content": "hello"}], "temperature": 0.9, "max_tokens": 1000, "stream": true, "model": "Qwen/Qwen2.5-0.5B-Instruct"}'
```

`transformers-serve` uses continuous batching (see this PR ↗ and also this one ↗) for better GPU utilization, and is very much linked to the great work of vLLM with the `paged attention kernel` — a further justification of external kernels.

`transformers-serve` is not meant for user-facing production services, tools like vLLM or SGLang are super optimized for that, but it's useful for several use cases:

- Quickly verify that your model is compatible with continuous batching and paged attention.
- Run ad-hoc vibe tests on any model, without worrying to deploy anything.

- Run evaluations efficiently, again without having to spend a lot of time engineering your infrastructure.

For model deployment, check Inference Providers ↗ or roll your solution using any of the excellent serving libraries.

> OpenAI-compatible surface + continuous batching; kernels/backends slot in because the modeling API stayed stable.
>
> Next: reuse across vLLM/SGLang relies on the same consistency.

Community reusability

The transformers-serve CLI is built on transformers, for sure, but the library is made first and foremost to be *reused* at large by the open-source ecosystem.

Adding a model to transformers means:

- having it immediately available to the community
- having it immediately usable in vLLM, SGLang ↗, and so on without additional code. In the case of vLLM, transformers was added as a backend to run models on vLLM, which optimizes throughput/latency on top of *existing* transformers architectures as seen in this great vLLM x HF blog post. ↗
- being the reference code for implementations in MLX, llama.cpp and other libraries.

This further cements the need for a consistent public surface : we are a backend and a reference, and there's more software than us to handle serving. At the time of writing, more effort is done in that direction. We already have compatible configs for VLMs for vLLM (say that three times fast), check here for GLM4 video support ↗, and here for MoE support ↗, for instance.

> Being a good backend consumer requires a consistent public surface; modular shards and configs make that stability practical.
>
> Next: what changes in v5 without breaking the promise of visible semantics.

A Pact with the Community and what is coming next

The next major version of `transformers` is just around the corner (and will have another blog post to its name when it comes out). When v5 is released, we aim to keep backwards compatibility as solid as possible. The changes we make now are in service of that goal.

We will lean further into a modular toolbox, not a framework. You should not be forced to rewrite modeling code. It's better when a model can inherit from `PreTrainedModel` and opt into Tensor Parallel, `from_pretrained`, sharding, `push_to_hub`, loss plumbing, and external stacks like PEFT/TRL/SGLang/vLLM.

We write this to make our design philosophy legible. Transformers is built by thousands of contributors, but it only stays usable if its core principles are explicit and upheld. These tenets are our pact with you: they ensure that whether you are shipping a new model, contributing an optimized kernel, or simply debugging a forward pass, the code remains transparent and hackable.

This is a living document, not a stone tablet. Tell us where these tenets fall short or should evolve next. We'll keep working, and we'll be here to share the journey with you all.

Citation

For attribution, cite this work as

```
Pablo Montalvo, Lysandre Debut, Pedro Cuenca, Yoni Gozlan (2025). "Maintain the unmaintainable:
1M python loc, 400+ models".
```

BibTeX citation

```
@misc{montalvo2025_maintain_the_unmaintaina,
  title={Maintain the unmaintainable: 1M python loc, 400+ models},
  author={Pablo Montalvo and Lysandre Debut and Pedro Cuenca and Yoni Gozlan},
  year={2025},

}
```


Acknowledgements

Special thanks to all the reviewers on this! Vaibhav Srivastav for his thoroughness, Cyril Vallez for his eagle eye, Yoni Gozlan (also for his excellent work on fast image processors), Arthur Zucker for his guidance, and of course the wonderful Thibaud Frere for designing this template and helping me out with it!

Most importantly: thanks to the entire Open-Source community, sincerely.