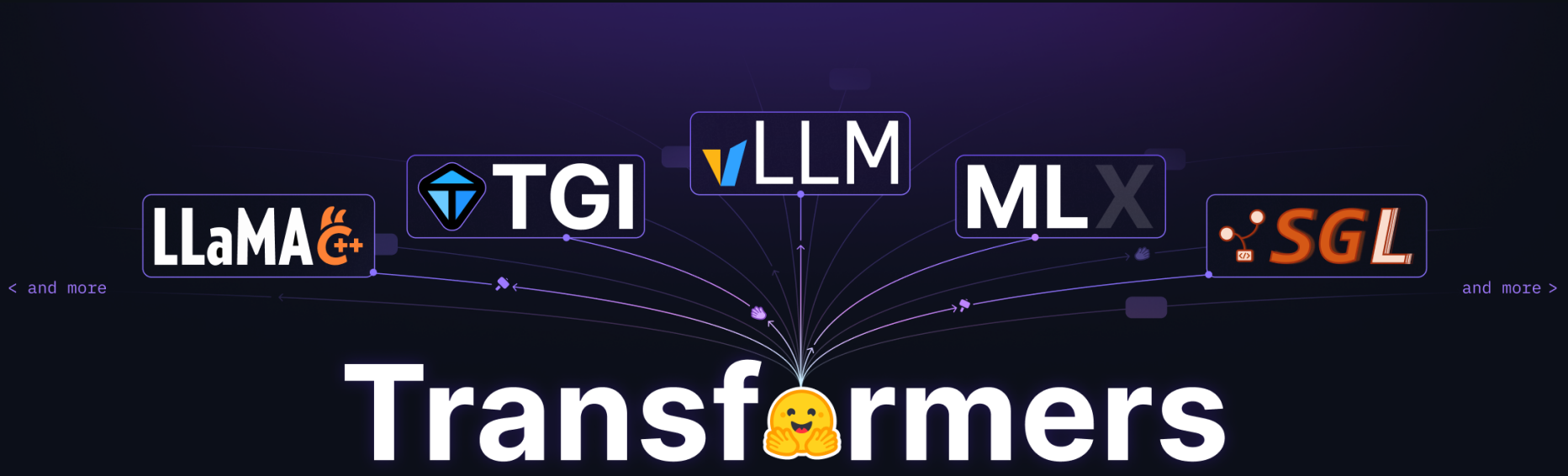
Write Once, Run Everywhere

with PyTorch Transformers

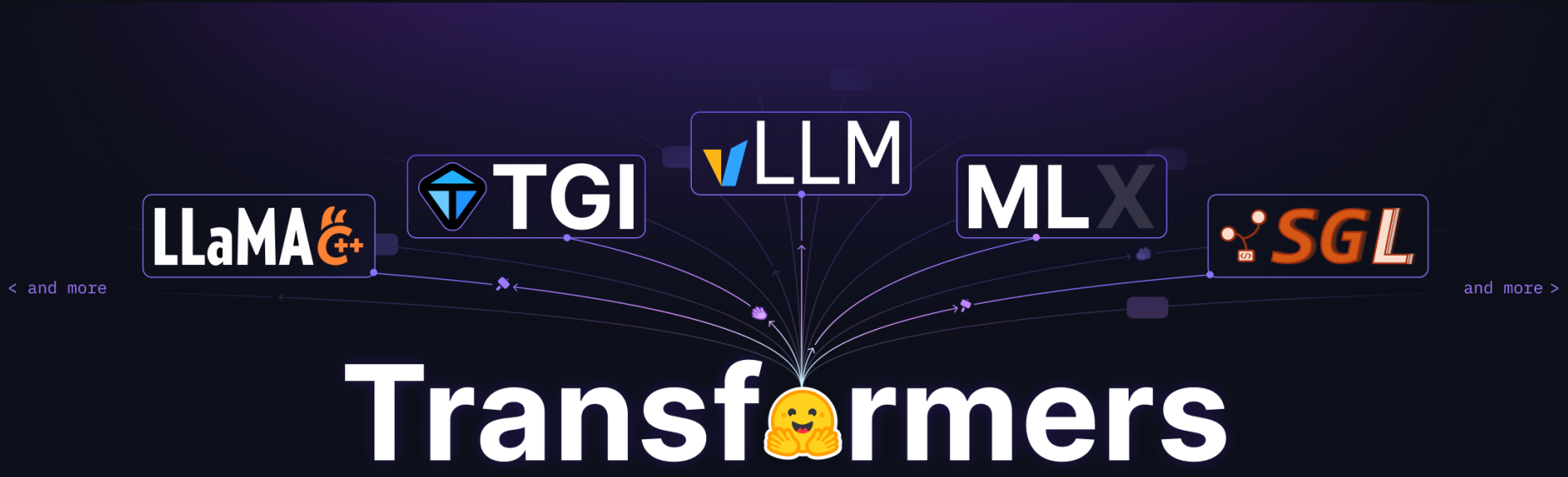
Pedro Cuenca

Hugging Face · PyTorch Conference Europe 2026

Transformers as the ML modeling reference



Transformers as the ML modeling reference



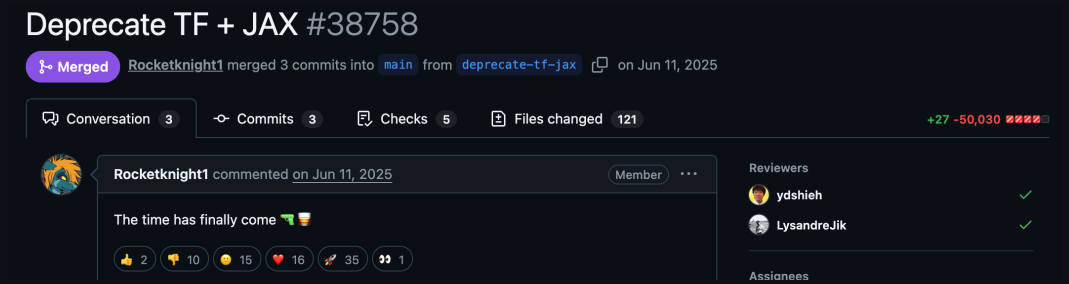
Goal: any model added to transformers becomes immediately available elsewhere

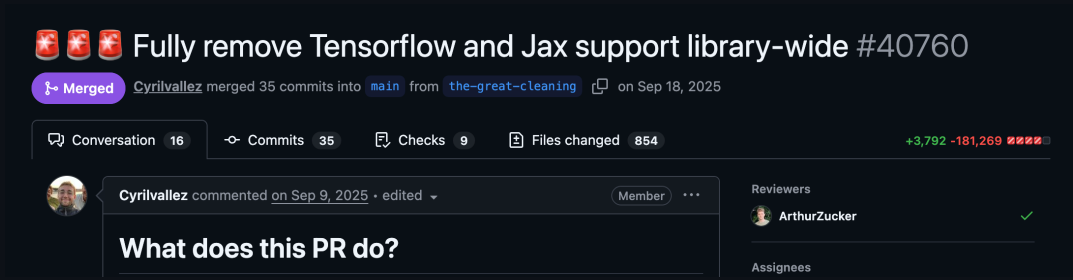
The Big Bet: PyTorch Only

231K
lines removed (without docs)

~1K
files changed

90%+
already on PyTorch





Velocity · Simplicity · Alignment

A Single Modeling Framework

PyTorch helps us build the abstractions that matter to us

A Single Modeling Framework

PyTorch helps us build the abstractions that matter to us

Modular Models

One model = one file
Contribute only the diffs

Easier contributions

A Single Modeling Framework

PyTorch helps us build the abstractions that matter to us

Modular Models

One model = one file
Contribute only the diffs

Attention Interface

Pluggable attention registry

Parallelism as Config

TP/PP plans in config
No modeling changes, no afterthought

Easier contributions

Sensible Abstractions by Standardization

torch.compile · FSDP · DTensor · FlexAttention · FlashAttention

A Single Modeling Framework

PyTorch helps us build the abstractions that matter to us

Modular Models

One model = one file
Contribute only the diffs

Attention Interface

Pluggable attention registry

Parallelism as Config

TP/PP plans in config
No modeling changes, no afterthought

Hub Kernels

Custom kernels downloadable from the Hub

Easier contributions

Sensible Abstractions by Standardization

Zero-install, opt-in, hardware dependent optimizations

torch.compile · FSDP · DTensor · FlexAttention · FlashAttention

Don’t miss Lysandre’s announcement tomorrow!

Hub Kernels

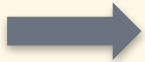
Custom kernels downloadable from the Hub

09:25 CEST	**Keynote: The Hub as Infrastructure. From Open PyTorch Models, to a Safe and Performant Distribution Hub - Lysandre Debut, Chief Open-Source Officer, Hugging Face** Master Stage

Zero-install, opt-in, hardware dependent optimizations

The Modular System

6,000+
lines per model (before)



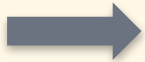
~200
lines of diffs (after)

tooling unravels the rest

```python
# modular_olmo2.py — contributor writes only the diffs

class Olmo2RMSNorm(LlamaRMSNorm):        # ← identical implementation, looks like inheritance
    pass

class Olmo2Attention(LlamaAttention):
    def __init__(self, config, layer_idx):
        super().__init__(config, layer_idx)
        self.q_norm = Olmo2RMSNorm(config.head_dim)   # ← the diff
        self.k_norm = Olmo2RMSNorm(config.head_dim)

# → Standalone modeling_olmo2.py unraveled from the modular file.
#   Readable top-to-bottom. No inheritance in the unraveled modeling file.
```

The Modular System

6,000+
lines per model (before)



~200
lines of diffs (after)

tooling unravels the rest

```python
# modular_olmo2.py — contributor writes only the diffs

class Olmo2RMSNorm(LlamaRMSNorm):        # ← identical implementation, looks like inheritance
    pass

class Olmo2Attention(LlamaAttention):
    def __init__(self, config, layer_idx):
        super().__init__(config, layer_idx)
        self.q_norm = Olmo2RMSNorm(config.head_dim)   # ← the diff
        self.k_norm = Olmo2RMSNorm(config.head_dim)

# → Standalone modeling_olmo2.py unraveled from the modular file.
#   Readable top-to-bottom. No inheritance in the unraveled modeling file.
```

one model == one file
Practitioners can understand reading top to bottom

Transformers ecosystem journey

Reference

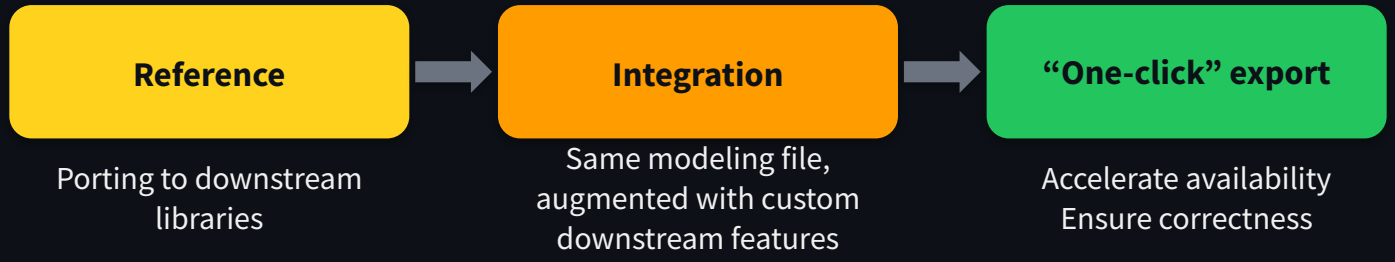
Porting to downstream libraries

Transformers ecosystem journey

Porting to downstream libraries

Same modeling file, augmented with custom downstream features

Transformers ecosystem journey



Transformers as the Reference Implementation

Downstream libraries typically start from the transformers implementation
The same code is ported to multiple frameworks

Can we do better?

No Porting Required

No Porting Required

vLLM

```
from vllm import LLM
llm = LLM(
    model="meta-llama/Llama-3.2-1B",
    model_impl="transformers"
)
```

No Porting Required

vLLM

```
from vllm import LLM
llm = LLM(
    model="meta-llama/Llama-3.2-1B",
    model_impl="transformers"
)
```

SGLang

```
import sglang as sgl
engine = sgl.Engine(
    "meta-llama/Llama-3.2-1B-Instruct",
    impl="transformers"
)
```

No Porting Required

vLLM

```python
from vllm import LLM
llm = LLM(
    model="meta-llama/Llama-3.2-1B",
    model_impl="transformers"
)
```

SGLang

```python
import sglang as sgl
engine = sgl.Engine(
    "meta-llama/Llama-3.2-1B-Instruct",
    impl="transformers"
)
```

```
$ m="llava-hf/llava-onevision-qwen2-0.5b-ov-hf"
$ vllm serve $f \
    --model_impl transformers \
```

No Porting Required

vLLM

```python
from vllm import LLM
llm = LLM(
    model="meta-llama/Llama-3.2-1B",
    model_impl="transformers"
)
```

```
$ m="llava-hf/llava-onevision-qwen2-0.5b-ov-hf"
$ vllm serve $f \
    --model_impl transformers \
```

SGLang

```python
import sglang as sgl
engine = sgl.Engine(
    "meta-llama/Llama-3.2-1B-Instruct",
    impl="transformers"
)
```

```
$ python3 -m sglang.launch_server \
    --model-path kyutai/helium-1-preview-2b \
    --impl transformers \
    --port 30000
```

No Porting Required

vLLM

```python
from vllm import LLM
llm = LLM(
    model="meta-llama/Llama-3.2-1B",
    model_impl="transformers"
)
```

```
$ m="llava-hf/llava-onevision-qwen2-0.5b-ov-hf"
$ vllm serve $f \
    --model_impl transformers \
```

SGLang

```python
import sglang as sgl
engine = sgl.Engine(
    "meta-llama/Llama-3.2-1B-Instruct",
    impl="transformers"
)
```

```
$ python3 -m sglang.launch_server \
    --model-path kyutai/helium-1-preview-2b \
    --impl transformers \
    --port 30000
```

Under the hood: PagedAttention, dynamic batching are “injected”

> *"The Transformers backend in vLLM has been very enabling to get more architectures available to more users. This is just the start of our collaboration."*
>
> — Simon Mo, Harry Mellor at vLLM

How It Works: AttentionInterface

Pluggable attention registry — built on PyTorch's extensibility

```python
from transformers import PreTrainedModel
from transformers.modeling_utils import ALL_ATTENTION_FUNCTIONS
from torch import nn


class MyAttention(nn.Module):
   def forward(self, hidden_states, **kwargs):
        ...
        attention_interface =
ALL_ATTENTION_FUNCTIONS[self.config._attn_implementation]
        attn_output, attn_weights = attention_interface(
            self, query_states, key_states, value_states, **kwargs
        )
        ...


class MyModel(PreTrainedModel):
    _supports_attention_backend = True
```

Model author writes:

Eager attention, once.
The registry dispatches to
the right backend at runtime.

Attention Backends:

PyTorch: SDPA · FlexAttention
Transformers: Flash Attention, Kernel Hub

vLLM injects: PagedAttention
SGLang injects: SGLang kernels

Standardized attention interface: Models ↔ Config ↔ Backends

How It Works: AttentionInterface

Pluggable attention registry — built on PyTorch's extensibility

```python
from transformers import PreTrainedModel
from transformers.modeling_utils import ALL_ATTENTION_FUNCTIONS
from torch import nn


class MyAttention(nn.Module):
    def forward(self, hidden_states, **kwargs):
        ...
        attention_interface = ALL_ATTENTION_FUNCTIONS[self.config._attn_implementation]
        attn_output, attn_weights = attention_interface(
            self, query_states, key_states, value_states, **kwargs
        )
        ...


class MyModel(PreTrainedModel):
    _supports_attention_backend = True
```

Model author writes:

Eager attention, once.
The registry dispatches to
the right backend at runtime.

Attention Backends:

PyTorch: SDPA · FlexAttention
Transformers: Flash Attention, Kernel Hub

vLLM injects: PagedAttention
SGLang injects: SGLang kernels

Standardized attention interface: Models ↔ Config ↔ Backends

How It Works: AttentionInterface

Pluggable attention registry — built on PyTorch's extensibility

```python
from transformers import PreTrainedModel
from transformers.modeling_utils import ALL_ATTENTION_FUNCTIONS
from torch import nn


class MyAttention(nn.Module):
    def forward(self, hidden_states, **kwargs):
        ...
        attention_interface = ALL_ATTENTION_FUNCTIONS[self.config._attn_implementation]
        attn_output, attn_weights = attention_interface(
            self, query_states, key_states, value_states, **kwargs
        )
        ...


class MyModel(PreTrainedModel):
    _supports_attention_backend = True
```

Model author writes:

Eager attention, once.
The registry dispatches to
the right backend at runtime.

Attention Backends:

PyTorch: SDPA · FlexAttention
Transformers: Flash Attention, Kernel Hub

vLLM injects: PagedAttention
SGLang injects: SGLang kernels

Standardized attention interface: Models ↔ Config ↔ Backends

Under the Hood

Before: per-model code

```python
# vLLM: custom LlamaAttention
class LlamaAttention(nn.Module):
    def __init__(self, ...):
        # vLLM-specific parallel layers
        self.qkv_proj = QKVParallelLinear(...)
        self.o_proj = RowParallelLinear(...)
        self.attn = Attention(...)

    def forward(self, h, pos, kv_cache):
        qkv, _ = self.qkv_proj(h)
        q, k, v = qkv.split(...)
        q, k = self.rotary_emb(pos, q, k)
        out = self.attn(q, k, v, kv_cache)
        output, _ = self.o_proj(out)
        return output
```

After: engine registers once, models use it

```python
# vLLM registers its attention ONCE:
ALL_ATTENTION_FUNCTIONS["vllm"] = vllm_attention

# Which can be plugged in a transformers model:
model = AutoModelForCausalLM.from_pretrained(
    model_id,
    attn_implementation="vllm",
)
```

Source: vllm/model_executor/models/llama.py

Under the Hood

Before: per-model code

```python
# vLLM: custom LlamaAttention
class LlamaAttention(nn.Module):
    def __init__(self, ...):
        # vLLM-specific parallel layers
        self.qkv_proj = QKVParallelLinear(...)
        self.o_proj = RowParallelLinear(...)
        self.attn = Attention(...)

    def forward(self, h, pos, kv_cache):
        qkv, _ = self.qkv_proj(h)
        q, k, v = qkv.split(...)
        q, k = self.rotary_emb(pos, q, k)
        out = self.attn(q, k, v, kv_cache)
        output, _ = self.o_proj(out)
        return output
```

After: engine registers once, models use it

```python
# vLLM registers its attention ONCE:
ALL_ATTENTION_FUNCTIONS["vllm"] = vllm_attention

# Which can be plugged in a transformers model:
model = AutoModelForCausalLM.from_pretrained(
    model_id,
    attn_implementation="vllm",
)
```

Source: vllm/model_executor/models/llama.py

Under the Hood

Before: per-model code

```
# vLLM: custom LlamaAttention
class LlamaAttention(nn.Module):
    def __init__(self, ...):
        # vLLM-specific parallel layers
        self.qkv_proj = QKVParallelLinear(...)
        self.o_proj = RowParallelLinear(...)
        self.attn = Attention(...)

    def forward(self, h, pos, kv_cache):
        qkv, _ = self.qkv_proj(h)
        q, k, v = qkv.split(...)
        q, k = self.rotary_emb(pos, q, k)
        out = self.attn(q, k, v, kv_cache)
        output, _ = self.o_proj(out)
        return output
```

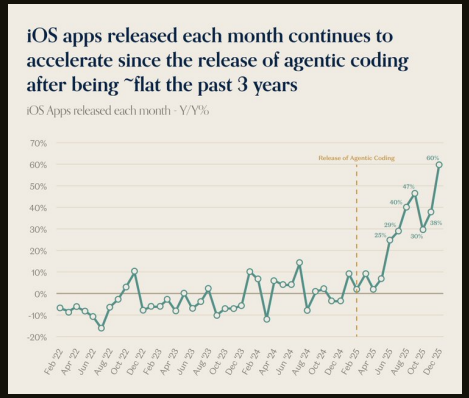
After: engine registers once, models use it

```
# vLLM registers its attention ONCE:
ALL_ATTENTION_FUNCTIONS["vllm"] = vllm_attention

# Which can be plugged in a transformers model:
model = AutoModelForCausalLM.from_pretrained(
    model_id,
    attn_implementation="vllm",
)
```

Source: vllm/model_executor/models/llama.py

Does this matter when agents can now write code?

Agents can now write code



a16z / Sensor Tower

Code = communication

Modeling code is written for humans.

Quality ≠ quantity

Agents miss implicit contracts, break APIs, try to adopt “best practices”.

Happening Everywhere

App Store, for instance. Coming to your repo soon.

It’s ok for many projects to ignore what code looks like.
But transformers is not one of them.

Learning to Work with Agents

Transformers release notes are LLM-generated

Lysandre Debut

✅ **More comprehensive**
LLM processes every commit exhaustively

✅ **Less bias**
No personal preference for the “latest” project or hottest topic

✅ **Higher quality**
Better grouping, highlights what matters to users

Contributing models with Codex

An experiment by Niels Rogge

Days ➡ Hours

~10 interactive prompts each to convert a model, guided by a domain expert

Structured guidance:
- AGENTS.md
- modular framework
- domain expertise

transformers-to-mlx Skill

transformers-to-mlx Skill

🎉 **Releasing today!**

transformers-to-mlx Skill

```
"Convert the olmo_hybrid architecture to MLX"
```

🎉 **Releasing today!**

transformers-to-mlx Skill

```
"Convert the olmo_hybrid architecture to MLX"
```

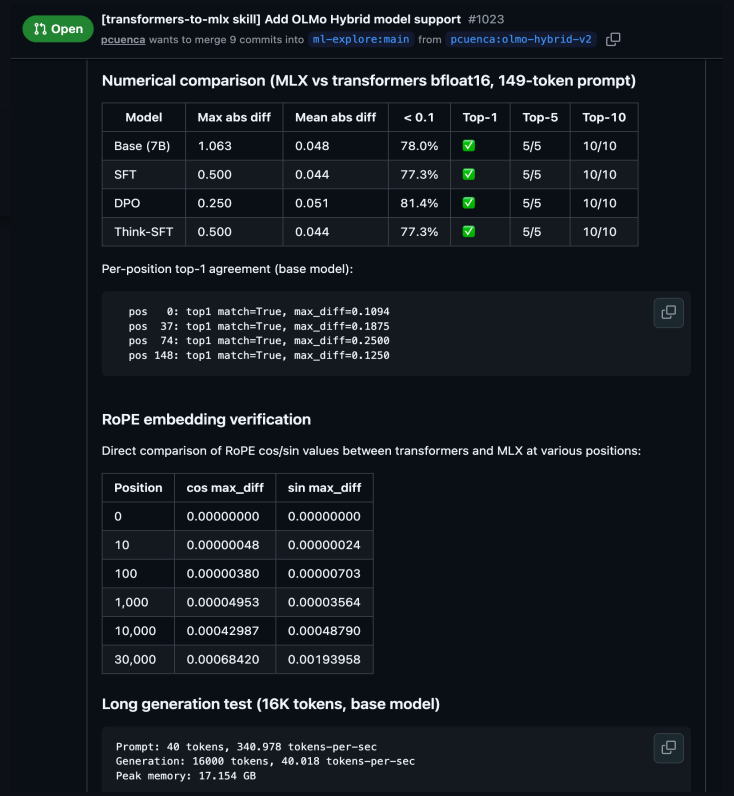
- ✅ Finds candidate models on the Hub
- ✅ Tests RoPE, dtypes, generation coherence
- ✅ Respects mlx-lm code style and idioms
- ✅ Long-sequence generation to uncover subtle bugs
- ✅ Generates comprehensive PR report
- ✅ Helper tool for contributor, who *owns* the PR
- ✅ Provides results and details to support the reviewer
- ✅ Non-agentic test harness for verification

🎉 **Releasing today!**

transformers-to-mlx Skill

"Convert the olmo_hybrid architecture to MLX"

- ✅ Finds candidate models on the Hub
- ✅ Tests RoPE, dtypes, generation coherence
- ✅ Respects mlx-lm code style and idioms
- ✅ Long-sequence generation to uncover subtle bugs
- ✅ Generates comprehensive PR report
- ✅ Helper tool for contributor, who *owns* the PR
- ✅ Provides results and details to support the reviewer
- ✅ Non-agentic test harness for verification



Open **[transformers-to-mlx skill] Add OLMo Hybrid model support** #1023

pcuenca wants to merge 9 commits into `ml-explore:main` from `pcuenca:olmo-hybrid-v2`

Numerical comparison (MLX vs transformers bfloat16, 149-token prompt)

Model	Max abs diff	Mean abs diff	< 0.1	Top-1	Top-5	Top-10
Base (7B)	1.063	0.048	78.0%	✅	5/5	10/10
SFT	0.500	0.044	77.3%	✅	5/5	10/10
DPO	0.250	0.051	81.4%	✅	5/5	10/10
Think-SFT	0.500	0.044	77.3%	✅	5/5	10/10

Per-position top-1 agreement (base model):

```
pos   0: top1 match=True, max_diff=0.1094
pos  37: top1 match=True, max_diff=0.1875
pos  74: top1 match=True, max_diff=0.2500
pos 148: top1 match=True, max_diff=0.1250
```

RoPE embedding verification

Direct comparison of RoPE cos/sin values between transformers and MLX at various positions:

Position	cos max_diff	sin max_diff
0	0.00000000	0.00000000
10	0.00000048	0.00000024
100	0.00000380	0.00000703
1,000	0.00004953	0.00003564
10,000	0.00042987	0.00048790
30,000	0.00068420	0.00193958

Long generation test (16K tokens, base model)

```
Prompt: 40 tokens, 340.978 tokens-per-sec
Generation: 16000 tokens, 40.018 tokens-per-sec
Peak memory: 17.154 GB
```

🎉 **Releasing today!**

Collaborating with Agents in Open Source

🛠️ **A tool for contributors**

🤝 **Supports reviewers**

✅ **Facilitates testing**

🎯 **Respects project idioms**

What's Next

mlx-vlm

Vision-language models
for Apple Silicon

llama.cpp

VLM pre-processing

One-click export

transformers-to-any

One model definition

Thank You!

Pedro Cuenca

@pcuenq · Hugging Face