isotope-typst

Building an English-math → Typst Transformer in Python with uv

A visual, step-by-step guide — from an empty folder to a model you can query.

The goal. A model that reads

```
the integral from 0 to infinity of e to the minus x squared dx
```

and writes `$integral_0^infinity e^(-x^2) dif x$`.

How to read this. Each section explains one stage, shows a diagram, gives you a small skeleton with `TODO`s, and ends with *Check yourself* questions. Try to answer them before peeking at Appendix A. The code is deliberately incomplete: the gaps are where the learning happens.

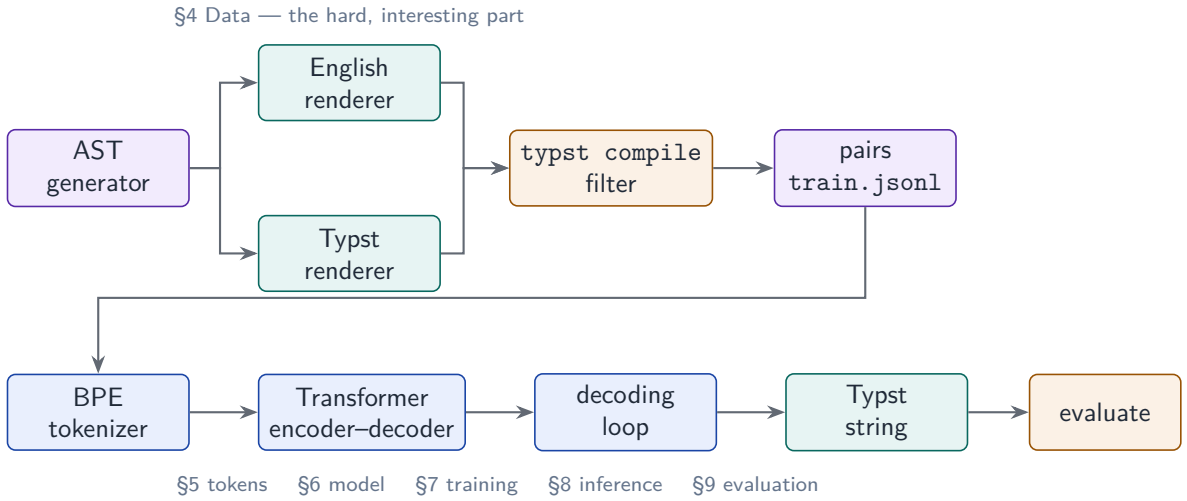


Figure 1: The whole pipeline. You will build it left-to-right, top-to-bottom.

1 What a Transformer is, in one page

A Transformer turns a sequence of tokens into a sequence of vectors, where every vector has "looked at" every other one through *attention*. For translation we use the original *encoder–decoder* shape (Vaswani et al., 2017):

- The **encoder** reads the whole English sentence at once and produces one vector per English token: a contextual "memory" of the input.
- The **decoder** writes Typst *one token at a time*. At each step it looks at (a) the Typst it has written so far, and (b) the encoder's memory, and predicts a probability distribution over the next token.

Attention itself is one formula. Each position produces a query Q, a key K and a value V:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}} + M\right) V$$

$QK^\top$ scores how relevant each position is to each other position; the softmax turns scores into weights; the result is a weighted average of values. M is a *mask* that adds $-\infty$ where looking is forbidden (padding, or the future).

Key idea

Your task is narrow: a small vocabulary of math words and a small, rigid output language. That is very good news. You do *not* need a 500M-parameter model to learn it. A ∼10M-parameter model trained from scratch on good synthetic data can get the synthetic test set nearly perfect. The hard part is making it robust to the way *real people* phrase math — and that is a data problem, not a model-size problem.

2 Pick your path (you can do both)

	Path A — from scratch	Path B — fine-tune pretrained
Model	Your own encoder–decoder, 2–50M params	ByT5-small (∼300M), or a small decoder-only LM (SmolLM2-360M, Qwen2.5-0.5B) with LoRA
What you learn	Every piece: tokenizer, attention, masks, training loop	Data formatting, LoRA, Hugging Face / MLX tooling
English robustness	Only what your data teaches it	Already "knows" English; generalises to odd phrasings much better
Training on M4 Air	Hours	Hours (LoRA) to a night
Size on disk	Tens of MB	Hundreds of MB to ∼1 GB

Recommendation. Build the *data generator and evaluation harness first* — they are shared by both paths and are 80% of the project. Then do Path A to understand Transformers, then Path B to see how much pretraining buys you, measured on the same test set. That comparison is also a nice little write-up.

Candid warning

Training a 100–500M model *from scratch* on a fanless MacBook Air is not a good plan: it needs far more data and compute than this task justifies, and the Air will thermally throttle on long runs. From scratch ⇒ stay small. Large ⇒ start from pretrained weights.

3 Set up the project with `uv`

```
uv init --package isotope-typst      # src/ layout, installable package
cd isotope-typst
uv python pin 3.12
uv add torch tokenizers tqdm         # Path A core
uv add --dev pytest ruff
brew install typst                   # the Typst CLI, used to validate data
uv run python -c "import torch; print(torch.backends.mps.is_available())"   # want: True
```

Suggested layout

```
isotope-typst/
  pyproject.toml
  uv.lock
  src/isotope_typst/
    ast.py          # math expression nodes
    render.py       # to_typst(), to_english()
    generate.py     # writes data/*.jsonl
    validate.py     # typst compile check
    tokenizer.py    # trains BPE
    model.py        # the Transformer
    train.py        # training loop
    translate.py    # inference / CLI
    evaluate.py     # metrics
  data/             # generated, git-ignored
  artifacts/        # tokenizer + checkpoints
  tests/
```

Expose commands in `pyproject.toml`

```
[project.scripts]
isotope-gen   = "isotope_typst.
    generate:main"
isotope-train = "isotope_typst.train:
    main"
isotope       = "isotope_typst.
    translate:main"
```

Then `uv run isotope-gen` etc. work, and `uv run pytest` runs tests. `uv.lock` pins every version so a run is reproducible on your Ubuntu box too.

4 Data: you have to manufacture it

There is no big public corpus of "spoken English math ↔ Typst". So you build a **generator**: create random math as a tree (an AST), then render that *same* tree two ways. The Typst side is deterministic; the English side is randomised across many phrasings. Because both come from one tree, every pair is correct by construction.

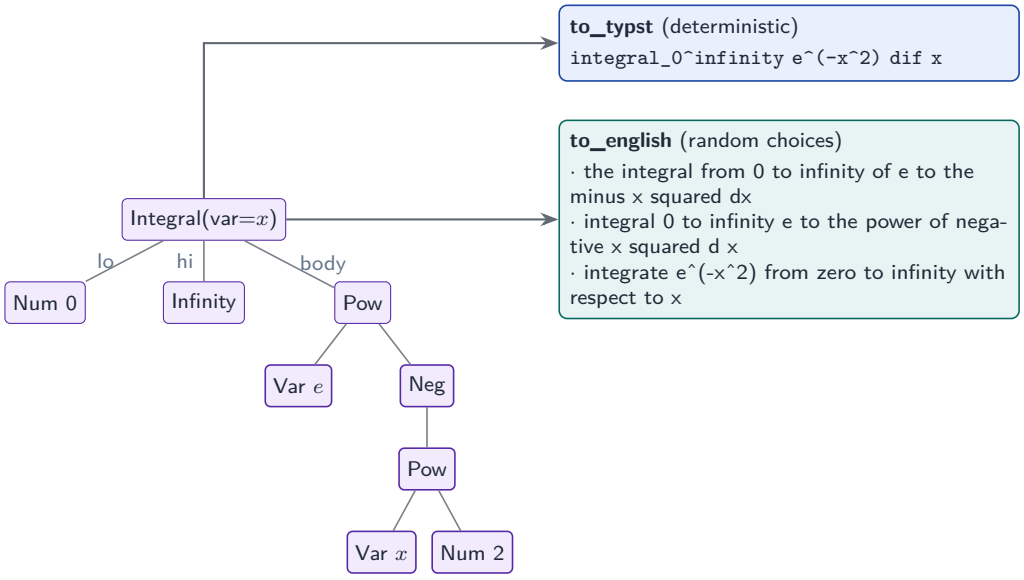


Figure 2: One tree, two renderings. Generate a million trees, get a million training pairs.

```
# src/isotope_typst/ast.py  +  render.py   (skeleton)
from dataclasses import dataclass
import random

class Node: ...

@dataclass
class Var(Node):  name: str
@dataclass
class Num(Node):  value: int
```

```
@dataclass
class Neg(Node):  arg: Node
@dataclass
class Pow(Node):  base: Node; exp: Node
@dataclass
class Integral(Node):
    body: Node; var: str; lo: Node | None = None; hi: Node | None = None
# TODO: Frac, Sqrt, Sum, Prod, Lim, Func("sin"|"ln"|...), Mul, Add, Sub, Eq, Greek vars

def group(n: Node) -> str:              # Typst needs parens around non-atomic sub/
    superscripts
    s = to_typst(n)
    return s if isinstance(n, (Var, Num)) else f"({s})"

def to_typst(n: Node) -> str:
    match n:
        case Var(name):  return name
        case Num(v):     return str(v)
        case Neg(a):     return f"-{to_typst(a)}"
        case Pow(b, e):  return f"{group(b)}^{group(e)}"
        case Integral(body, v, lo, hi):
            lim = f"_{group(lo)}^{group(hi)}" if lo is not None else ""
            return f"integral{lim} {to_typst(body)} dif {v}"
    raise NotImplementedError(n)

def to_english(n: Node, rng: random.Random) -> str:
    match n:
        case Pow(b, Num(2)):
            base = to_english(b, rng)
            return rng.choice([f"{base} squared", f"{base} to the power of 2",
                               f"{base} to the second"])
        # TODO: every node type gets 2-6 phrasings, chosen at random
    raise NotImplementedError(n)
```

4.1 Making the data good (this is where the model's quality comes from)

- **Vary the English hard.** Synonyms (*minus / negative*), number words (*two / 2*), optional *the*, *dx / d x / with respect to x*, spelled-out Greek, casual forms (*x over 2*, *half of x*). Add noise: lowercase/uppercase, dropped words, the odd typo.
- **Control the distribution.** Sample tree depth from a range (mostly shallow, some deep). Log a histogram of node types — if `Lim` is 0.3% of data, it will be bad at limits.
- **Validate the Typst side.** Compile every distinct Typst string once; drop failures. A broken renderer silently teaches the model broken Typst.
- **Split by structure, not by row.** If the same tree appears in train and test with different English, your test score is inflated. Hash the Typst string and split on the hash.
- **Write a real test set by hand.** 150–300 sentences *you* (and a few friends) write naturally, with their correct Typst. Never generate it, never train on it.
- **Later: paraphrase with a local LLM.** Feed generated English to a local model and ask for rephrasings; keep the Typst target unchanged. This is the cheapest robustness boost.

```python
# src/isotope_typst/validate.py  -- simple version; batch many equations per file for speed
import pathlib, subprocess, tempfile

def compiles(math: str) -> bool:
    with tempfile.TemporaryDirectory() as d:
        src = pathlib.Path(d) / "t.typ"
        src.write_text(f"${math}$\n")
        r = subprocess.run(["typst", "compile", str(src)], capture_output=True)
        return r.returncode == 0
```

Your turn

Implement `Num`, `Var`, `Neg`, `Pow`, `Frac` and `Integral` only. Generate 10,000 pairs into `data/train.jsonl` as `\{"en": ..., "typ": ...\}` lines. Every Typst string must compile. Then write 20 hand-made test sentences. Don't add more node types until the whole pipeline works end-to-end.

Check yourself 1

1. *"e to the minus x squared"* — is that e^{-x^2} or $(e^{-x})^2$? What should your generator do about sentences that are genuinely ambiguous?
2. Why must the hand-written test set never come from your generator?

5 Tokenization: text → integers

The model sees integer IDs, not characters. A byte-level **BPE** tokenizer learns frequent chunks from your data (`integral`, `squared`, `dif`) and can still spell anything byte-by-byte. Train *one* tokenizer on both English and Typst.

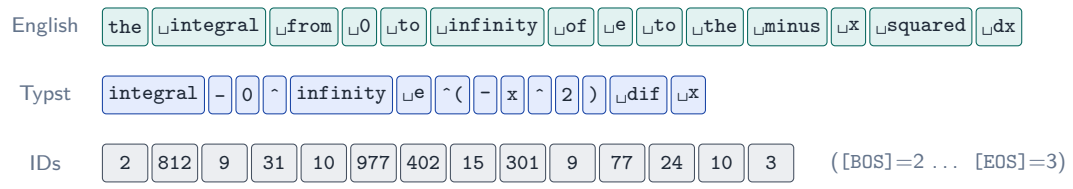


Figure 3: Illustrative split — your BPE will choose its own chunks. ␣ marks a leading space.

```python
# src/isotope_typst/tokenizer.py
from tokenizers import Tokenizer, models, trainers, pre_tokenizers, decoders

SPECIALS = ["[PAD]", "[UNK]", "[BOS]", "[EOS]"]       # ids 0, 1, 2, 3

def train(files: list[str], out: str = "artifacts/tokenizer.json", vocab: int = 4000):
    tok = Tokenizer(models.BPE(unk_token="[UNK]"))
    tok.pre_tokenizer = pre_tokenizers.ByteLevel(add_prefix_space=False)
    tok.decoder = decoders.ByteLevel()
    trainer = trainers.BpeTrainer(vocab_size=vocab, special_tokens=SPECIALS,
                                  initial_alphabet=pre_tokenizers.ByteLevel.alphabet())
    tok.train(files, trainer)            # files: plain-text dumps of the en and typ columns
    tok.save(out)
```

Check yourself 2

3. Why share one tokenizer between English and Typst instead of training two?

Test to write: for 1,000 random Typst strings, `decode(encode(s)) == s`. If round-tripping fails, nothing downstream can be right.

6 The model

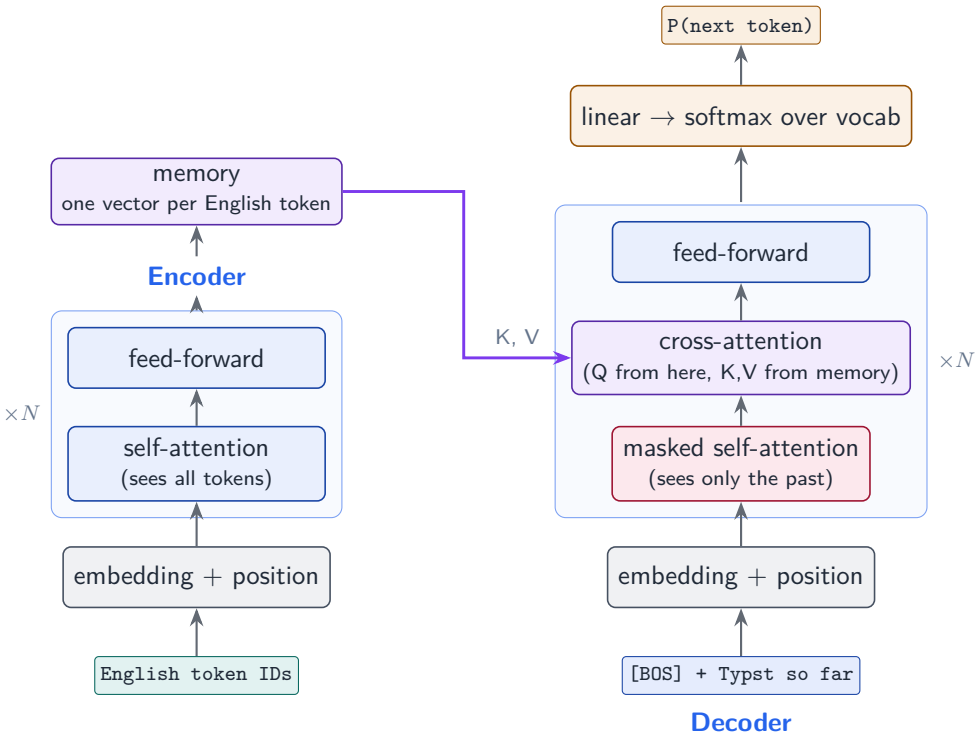


Figure 4: Encoder–decoder Transformer. Pre-norm and residual connections around every sub-block are omitted for clarity.

6.1 Masks: what may look at what

Two masks. **Padding mask**: batches pad short sequences with `[PAD]`; nobody may attend to those. **Causal mask**: during training the decoder gets the entire target at once (for speed), so position i must be blocked from seeing positions $> i$, or it would just copy the answer.

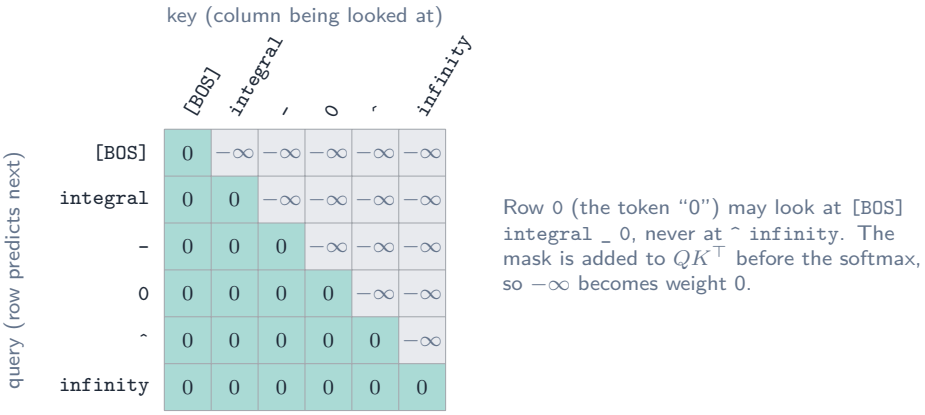


Figure 5: The causal mask M for a 6-token target.

```python
# src/isotope_typst/model.py   (Path A, using PyTorch's built-in layers)
import torch, torch.nn as nn

class Seq2Seq(nn.Module):
    def __init__(self, vocab, d=256, heads=4, layers=4, ff=1024, max_len=256, drop=0.1):
        super().__init__()
        self.tok_emb = nn.Embedding(vocab, d)
        self.pos_emb = nn.Embedding(max_len, d)          # learned positions: simplest
    option
        self.tf = nn.Transformer(d, heads, layers, layers, ff, drop,
                                 batch_first=True, norm_first=True)
        self.out = nn.Linear(d, vocab, bias=False)
        self.out.weight = self.tok_emb.weight             # weight tying (shared
    tokenizer!)

    def emb(self, ids):
        pos = torch.arange(ids.size(1), device=ids.device)
        return self.tok_emb(ids) + self.pos_emb(pos)

    def forward(self, src, tgt_in, pad_id=0):
        T = tgt_in.size(1)
        causal = torch.triu(torch.ones(T, T, dtype=torch.bool, device=src.device), 1)
        h = self.tf(self.emb(src), self.emb(tgt_in), tgt_mask=causal,
                    src_key_padding_mask=(src == pad_id),
                    tgt_key_padding_mask=(tgt_in == pad_id),
                    memory_key_padding_mask=(src == pad_id))
        return self.out(h)                                # (batch, T, vocab) logits

    # TODO: encode(src) -> memory  and  decode(tgt_in, memory) -> logits,
    #       using self.tf.encoder / self.tf.decoder. You need them for inference.
```

Your turn

Once the built-in version trains, write your own `MultiHeadAttention` module (about 30 lines: project to Q, K, V, reshape into heads, apply the formula, merge heads, project out) and swap it in. Check it against `nn.MultiheadAttention` on the same random input. That exercise is where attention "clicks".

6.2 How big?

Per layer, roughly: encoder $4d^2 + 2\,d\,d_{\text{ff}}$, decoder $8d^2 + 2\,d\,d_{\text{ff}}$ (why 8? count the attention blocks), plus $|V| \cdot d$ for the tied embedding.

config	d	layers (enc+dec)	heads	d_{ff}	params ($\lvert V\rvert = 4$k)
tiny (debugging)	128	3+3	4	512	$\approx$ 2M
small (start here)	256	4+4	4	1024	$\approx$ 8M
medium	512	6+6	8	2048	$\approx$ 46M

Check yourself 3

4. What would happen to training loss, and to real outputs, if you forgot the causal mask?
5. In cross-attention, why do queries come from the decoder but keys and values from the encoder?

7 Training

Teacher forcing. The decoder input is the target shifted right by one; the label at each position is the next token. One forward pass trains every position in parallel.

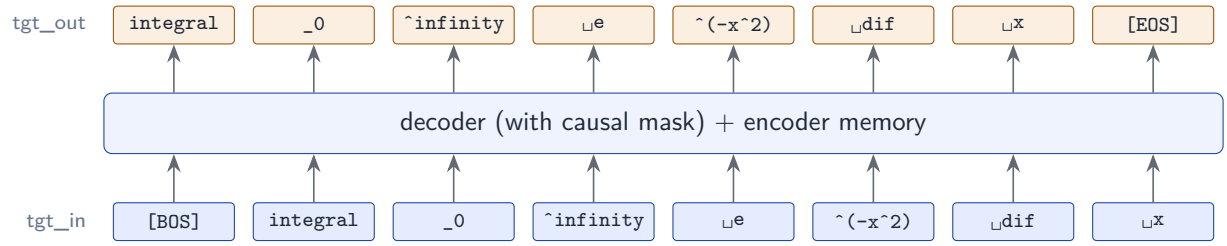
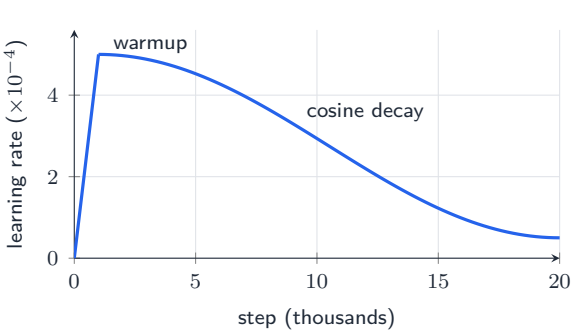


Figure 6: `tgt_in = tgt[:, :-1]`, `tgt_out = tgt[:, 1:]`. Loss = cross-entropy at every orange box.

```
# src/isotope_typst/train.py  (core of the loop)
import torch, torch.nn.functional as F
dev = "mps" if torch.backends.mps.is_available() else "cpu"
model = Seq2Seq(vocab=V).to(dev)
opt = torch.optim.AdamW(model.parameters(), lr=5e-4, weight_decay=0.01)
sched = ...   # TODO: linear warmup then cosine decay (see plot)

for step, (src, tgt) in enumerate(loader):            # tgt = [BOS] y1 ... yn [EOS], padded
    src, tgt = src.to(dev), tgt.to(dev)
    tgt_in, tgt_out = tgt[:, :-1], tgt[:, 1:]          # teacher-forcing shift
    logits = model(src, tgt_in)
    loss = F.cross_entropy(logits.reshape(-1, V), tgt_out.reshape(-1),
                           ignore_index=PAD, label_smoothing=0.1)
    opt.zero_grad(set_to_none=True)
    loss.backward()
    torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
    opt.step(); sched.step()
    # TODO: every N steps -> val loss, 20 sample translations printed, save checkpoint
```



Practical notes for the M4 Air

- Use `device="mps"`. Keep batches of similar length together (sort by length into buckets) — padding is wasted compute.
- Checkpoint often; a long run will throttle and you may want to stop it.
- Log to a CSV and plot it, or `uv add tensorboard`.
- The **tiny** config should train in minutes. Use it for every bug hunt.

Key idea

Before training on everything, overfit 100 examples. Train the tiny model on 100 pairs until loss is ≈ 0 and greedy decoding reproduces them exactly. If it cannot memorise 100 examples, your pipeline has a bug (shift, mask, special tokens), and more data will not fix it.

Check yourself 4

6. Training loss is low, but greedy outputs at inference are garbage. List three likely causes.

8 Inference: writing one token at a time

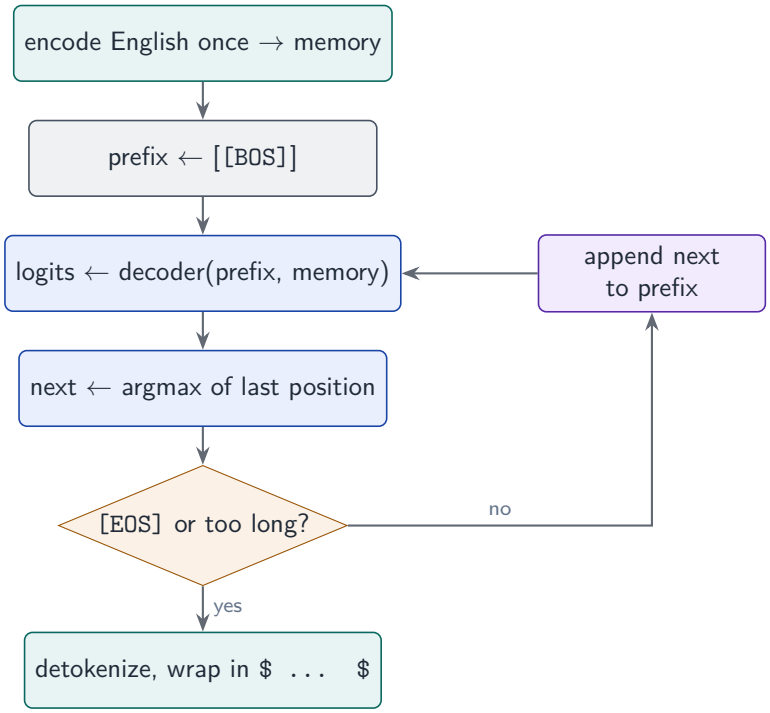


Figure 7: Greedy decoding. *Beam search* keeps the k best prefixes instead of one — try it once greedy works.

```python
@torch.no_grad()
def translate(model, tok, text: str, max_len: int = 128) -> str:
    model.eval()
    src = torch.tensor([tok.encode(text).ids], device=dev)
    memory = model.encode(src)
    out = [BOS]
    for _ in range(max_len):
        logits = model.decode(torch.tensor([out], device=dev), memory)
        nxt = int(logits[0, -1].argmax())
        if nxt == EOS:
            break
        out.append(nxt)
    return tok.decode(out[1:])
```

A neat extra later: **constrained decoding**. Since you know Typst's grammar, you can forbid tokens that would make the output unbalanced (e.g. a `)` with no open `(`). Cheap, and it removes a whole class of errors.

9 Evaluation: measure what you actually care about

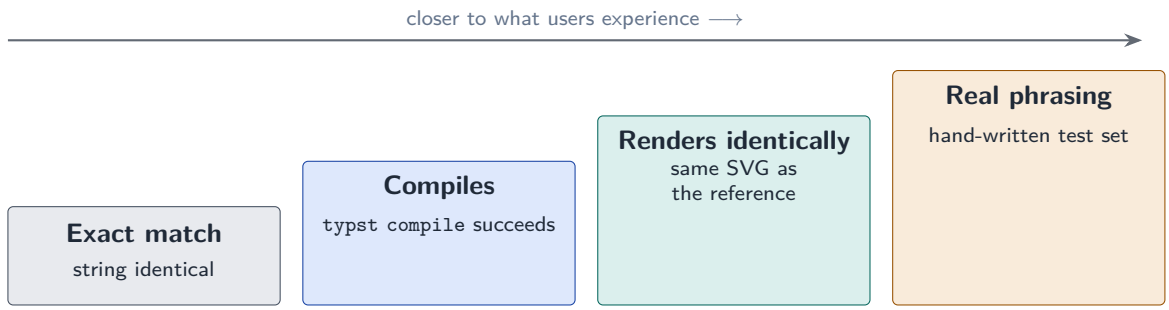


Figure 8: An evaluation ladder. Report all four, on the synthetic val set *and* the hand-written set.

- **Exact match** is easy but too strict: `x^2` and `x^(2)` render the same.
- **Compile rate** catches syntactically broken output.
- **Render match**: compile both prediction and reference to SVG (`typst compile t.typ t.svg`) and compare. This is the closest automatic metric to "correct".
- **Error analysis beats metrics**: dump the 50 worst hand-test failures and group them. Each group usually points at a missing generator phrasing.

Check yourself 5

7. Your synthetic val accuracy is 99%, your hand-written test accuracy is 40%. What does that tell you, and what do you change — the model or the data?

10 Path B: fine-tuning a pretrained model

Same data, same evaluation, different model. Two good options:

(1) Seq2seq: ByT5-small via Hugging Face `transformers` (`uv add transformers datasets accelerate`). It works on raw bytes, so every Typst character is representable — worth checking, because some older vocabularies (original T5) cannot encode curly braces at all. Use `Seq2SeqTrainer` with `en` as input and `typ` as label.

(2) Decoder-only + LoRA, e.g. SmolLM2-360M or Qwen2.5-0.5B. On Apple Silicon the most comfortable tool is `mlx-lm` (`uv add mlx-lm`), whose LoRA trainer reads a folder with `train.jsonl` / `valid.jsonl`. Turn each pair into a prompt and a completion:

```
{"prompt": "Translate to Typst math:\nthe integral from 0 to infinity of e to the minus
    x squared dx\nTypst:",
 "completion": " integral_0^infinity e^(-x^2) dif x"}
```

Check the current `mlx-lm` README for exact flags; they change between releases.

Key idea

Pretraining is why Path B handles *"integrate e to the negative x squared, zero to infinity"* even if your generator never produced that word order. Your Path A model only knows the English you taught it. The gap between the two on the hand-written set is the measure of what pretraining is worth here.

11 Roadmap

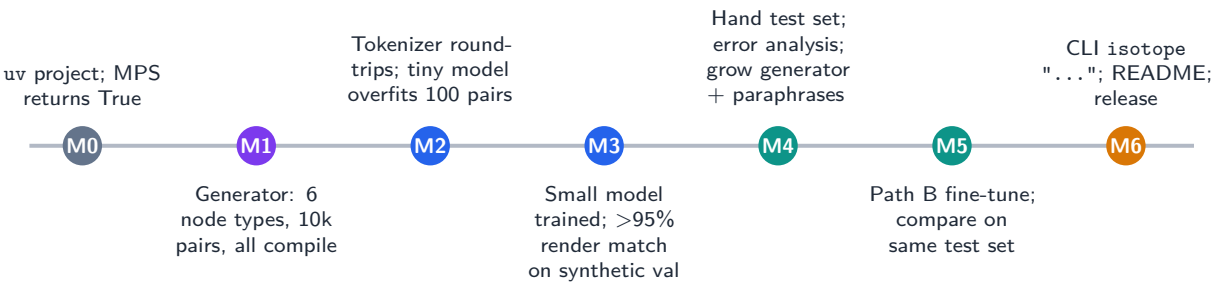


Figure 9: Each milestone has a "done when" test. Don't move on until it passes.

Order of work, honestly: M1 and M4 (data) will take more time than everything else combined. That is normal; it is also the part that makes the model good.

A Answers to the check-yourself questions

1. Spoken math is ambiguous. Pick a *convention* (here: "e to the ..." takes everything after it as the exponent, so e^{-x^2}) and make the generator produce *only* unambiguous or convention-following English for each tree. To express $(e^{-x})^2$ it must say something different, e.g. "e to the minus x, all squared". Inconsistent data teaches the model to guess.
2. A generated test set only measures how well the model learned *your templates*. The question you care about is whether it understands people, so the test must come from people.
3. Many tokens appear on both sides (digits, variable names, `x`, `pi`). One vocabulary lets the model copy them and lets you tie input embeddings, output embeddings and the output projection — fewer parameters, faster learning.
4. Training loss collapses toward zero because each position can see the token it is supposed to predict. At inference there is no future to peek at, so outputs are nonsense.
5. The decoder is the one asking "which English words matter for the token I'm writing now?" — that is a query. The English is what is being searched (keys) and what information is retrieved (values).
6. (a) Shift mismatch between training and inference; (b) causal mask missing or inverted (`True` means *blocked* in PyTorch bool masks); (c) `[BOS]`/`[EOS]` not added the same way in both places; also: forgetting `model.eval()` (dropout on), or different tokenizer files.

7. Classic overfitting to the generator's distribution. The model is fine; the *data* is too narrow. Look at the failures, add phrasings and paraphrase augmentation, regenerate, retrain.

Glossary. AST — abstract syntax tree, the structured form of an expression. **BPE** — byte-pair encoding, a tokenizer that merges frequent byte pairs into tokens. **Logits** — unnormalised scores before softmax. **Teacher forcing** — feeding the true previous tokens during training. **LoRA** — training small low-rank adapter matrices instead of all weights. **MPS** — PyTorch's Apple-GPU backend.