Build a Language Model on Your Mac

Flavio Copes

Updated August 3, 2026

Contents

Use Language Model Builder to design, train, inspect, fine-tune, and evaluate a small language model entirely on your Mac.

What you'll learn: Complete one documented language-model experiment from blueprint and data selection through pre-training, fine-tuning, comparison, and an honest lab report.

Plan the experiment

Install the app, set realistic expectations, and define what you will measure.

What we are building

Define one small language-model experiment you can train, inspect, compare, and explain without pretending it is a frontier model.

We are going to build a small language model on your Mac with Language Model Builder.

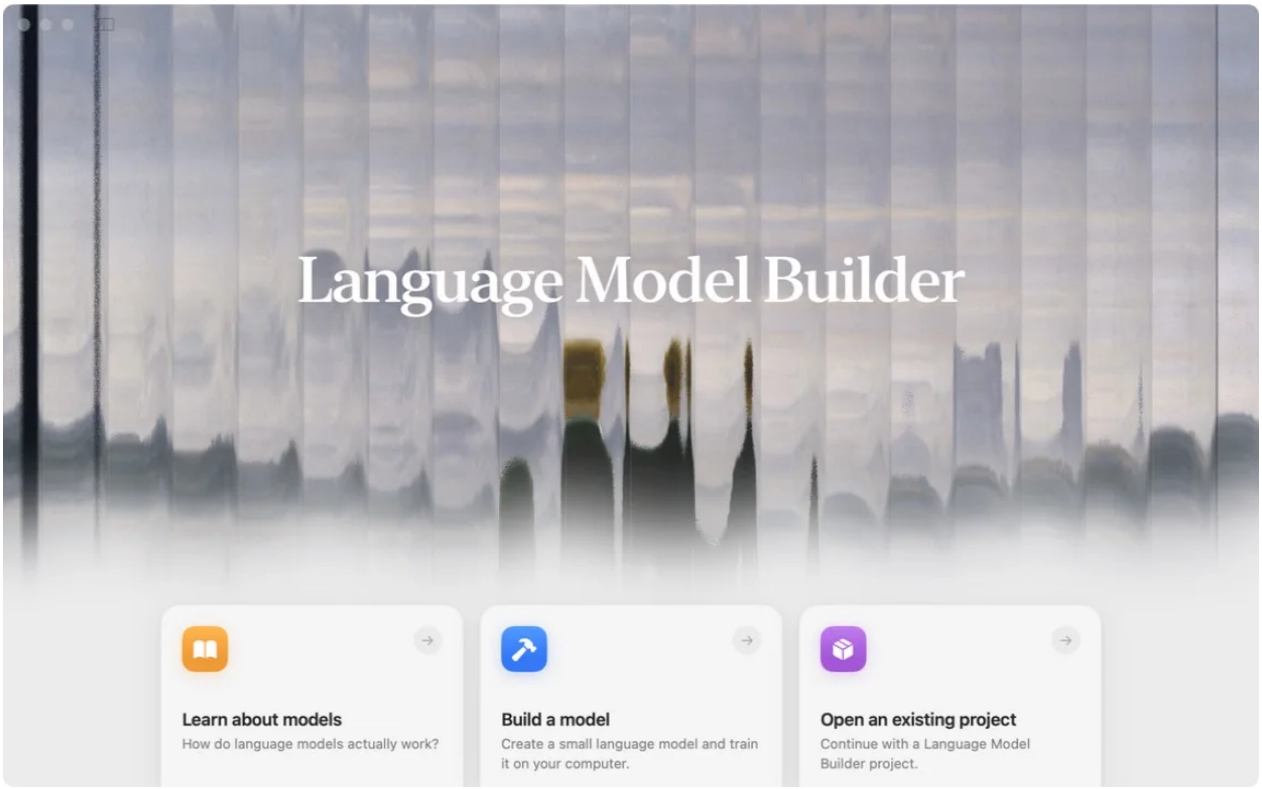
The goal is not to replace ChatGPT or Claude. The goal is to explain one complete learning system with evidence you collected yourself.

Our experiment follows this chain:

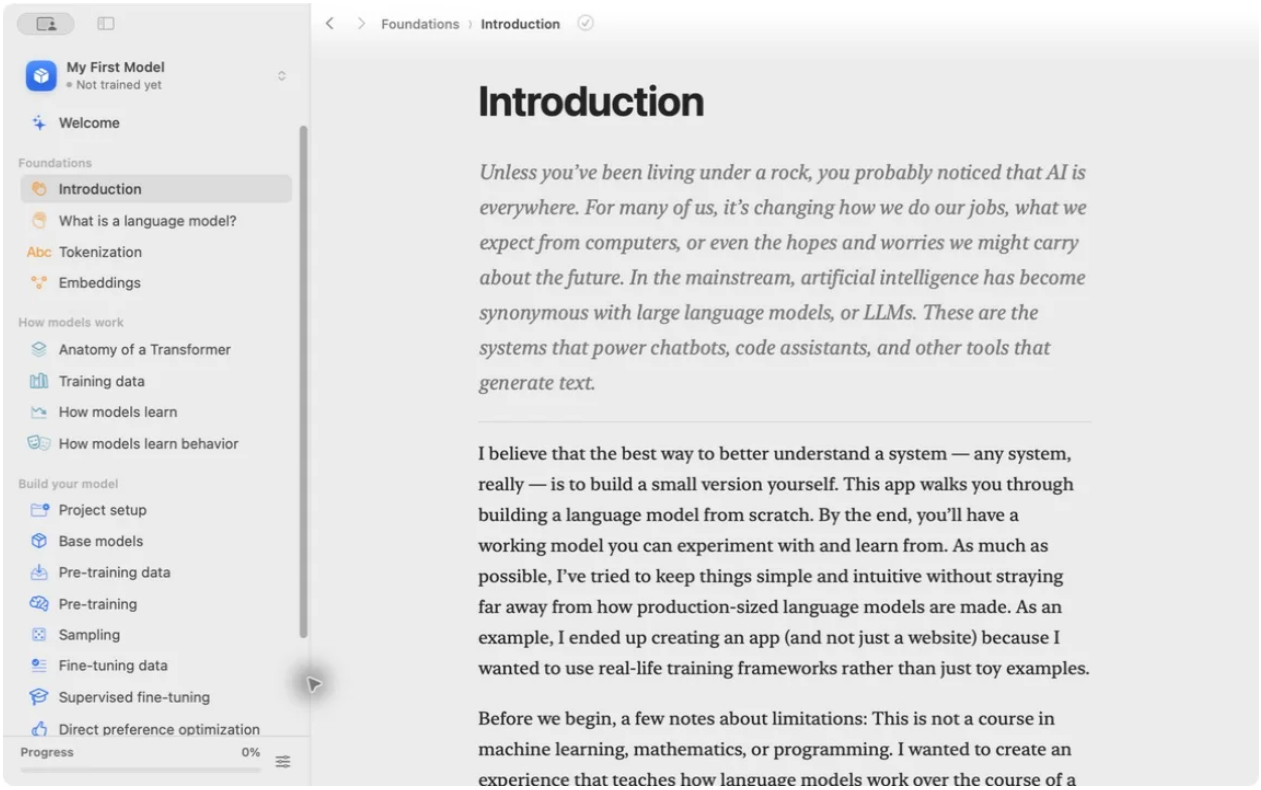
```
text → tokens → training examples → next-token loss
     → updated parameters → checkpoints → sampled text → evaluation
```

Later, supervised fine-tuning and preference training will shape how the same model responds. They do not replace tokenization, pre-training, or evaluation.

Open the app and choose **Build a model**. Do not start training yet.



The built-in textbook is useful when you want to pause the lab and revisit one idea without leaving the app.



Create a new project with a simple name such as `Tiny Stories Lab`. The name should help you identify this experiment later.

Write one hypothesis before changing a setting:

> A small model trained on short stories will move from random tokens toward grammatical story continuations, but it will remain unreliable outside that domain.

By the end, you will have:

- the exact tokenizer, data split, and model blueprint
- training and validation curves
- samples from early, middle, and late checkpoints
- SFT and preference comparisons
- a report separating observations from explanations

The report matters as much as the weights. A model file cannot tell us which data it saw, which controls stayed fixed, or why we trust a conclusion.

Install the app and check your Mac

Confirm the current macOS and Apple Silicon requirements, install the app from its official source, and keep enough local space available.

Language Model Builder currently requires an Apple Silicon Mac and macOS 15 or later. This course was checked with version 1.2.2. Verify both details on the official download page because application requirements can change.

Choose **About This Mac** from the Apple menu. Confirm the chip starts with `Apple M` and the macOS version is 15 or later.

Download the notarized app from the official website or its GitHub releases. Do not download a copy from an unrelated mirror.

Keep enough free disk space for the app, datasets, checkpoints, and exported weights. A dataset download can be much larger than the final model file.

Training uses unified memory shared by the CPU and GPU. Close memory-heavy applications before a long run, but record that choice so timing comparisons stay fair.

Create a short environment record:

```
Mac chip:
Unified memory:
macOS version:
App version:
Free disk space before the run:
```

Open the app once and create the project from the previous lesson. If macOS blocks it, verify that you used the official build before changing any security setting.

The app keeps projects, datasets, checkpoints, prompts, and training results locally. Update checks and optional crash reporting have separate network behavior, so review the current privacy settings if the dataset is sensitive.

After creating the project, quit and reopen the app. Confirm the empty project remains available before committing hours to training.

What to observe: the project appears in the sidebar and shows that it has not been trained yet.

What a small model can and cannot do

Set useful expectations for an educational model and choose observations that its scale can genuinely support.

A small model can learn token patterns, spelling, sentence shapes, recurring names, and simple relationships present in its data.

It can also produce broken grammar, repeat itself, contradict the prompt, or invent details. These failures are part of the experiment.

Do not evaluate it as a general assistant. Ask whether it improves from noise toward the distribution it trained on.

Choose three baseline prompts now. For a story dataset, you might use:

- `Once upon a time, a small fox`
- `The robot opened the door and`
- `Mia looked at the sky because`

Save those exact prompts. We will reuse them at several checkpoints with the same sampling settings.

Add a small rubric before seeing any trained output:

Check	Score
Forms recognizable words	0 or 1
Completes a sentence	0 or 1
Stays on the prompt topic	0 or 1
Avoids obvious repetition	0 or 1

This rubric is intentionally modest. "Factually correct" is a poor primary target for a tiny story model with narrow data.

Generate from the untrained model if the app allows it. Save the raw output, checkpoint state, prompt, sampling settings, and seed. Do not keep only the funniest failure.

Our later claim will be narrow: under fixed generation settings, the trained checkpoints score better on this defined story rubric than the initial state.

What to observe: before training, output should be random or unusable. That gives us a real baseline.

Start a model lab notebook

Create a simple record for configuration, data, observations, outputs, and decisions before the first training run begins.

Create a Markdown file named `model-lab.md`, or use any notes app you trust. This will become the experiment record.

Start with these headings:

```
# Tiny Stories Lab

## Goal
## Mac and app version
## Run manifest
## Model configuration
## Tokenizer and dataset
## Data split
## Training log
## Checkpoint samples
## Evaluation settings
## Fine-tuning
## Limitations
```

Write the goal in one sentence. Example: "Observe how a small transformer changes during pretraining, then test whether SFT improves instruction-shaped responses."

Record your Mac chip, memory, macOS version, and app version. These details help explain timing differences.

Paste the three baseline prompts from the previous lesson.

Create a run manifest before training:

```
Run ID: starter-stories-v1
Random seed:
Tokenizer and vocabulary size:
Training, validation, and test data IDs:
Context length:
Model shape:
Batch size:
Step budget:
Learning-rate settings:
```

```
Sampling settings:
```

Record units. “Memory: 18” and “speed: 900” are useless without MB, GB, tokens per second, or another defined unit.

Keep observations separate from explanations:

```
Observation: validation loss rose after step 12,000.
Possible explanation: the model began fitting training-specific patterns.
Next test: compare the 10,000 and 12,000 checkpoints under fixed sampling.
```

Never edit an old result to match a later interpretation. Add a dated correction instead.

What to observe: the notebook begins with facts and questions, not a conclusion written in advance.

Check your understanding: plan the experiment

Check the app requirements, realistic small-model goals, baseline records, and how a long-running training experiment fits the course.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

See what the model sees

Inspect tokens, embeddings, attention, and the transformer path.

Tokens and token IDs

Turn text into the numbered pieces a language model receives and notice that tokens are not the same as words.

A model does not receive a string or a list of English words. A **tokenizer** converts text into pieces and maps each piece to an integer ID.

Imagine this tiny vocabulary:

```
0: <end>
1: " the"
2: " cat"
3: " sat"
4: "."
```

The text `the cat sat.` could become `[1, 2, 3, 4]`. The model sees those IDs, then looks up a learned vector for each one.

Open **Tokenization** in the app. Enter one of your baseline prompts and inspect the token stream.

Try a common word, an unusual name, punctuation, and an emoji. Record which inputs use one token and which split into several pieces.

Check round-trip behavior too:

```
decode(encode(text)) = text
```

If normalization changes spaces, punctuation, or Unicode, document it. The tokenizer defines the exact text the model can reconstruct.

The model predicts one next token from its vocabulary. It does not first decide on a complete sentence.

A token can include a leading space or only part of a word. That is why token counts do not match word counts.

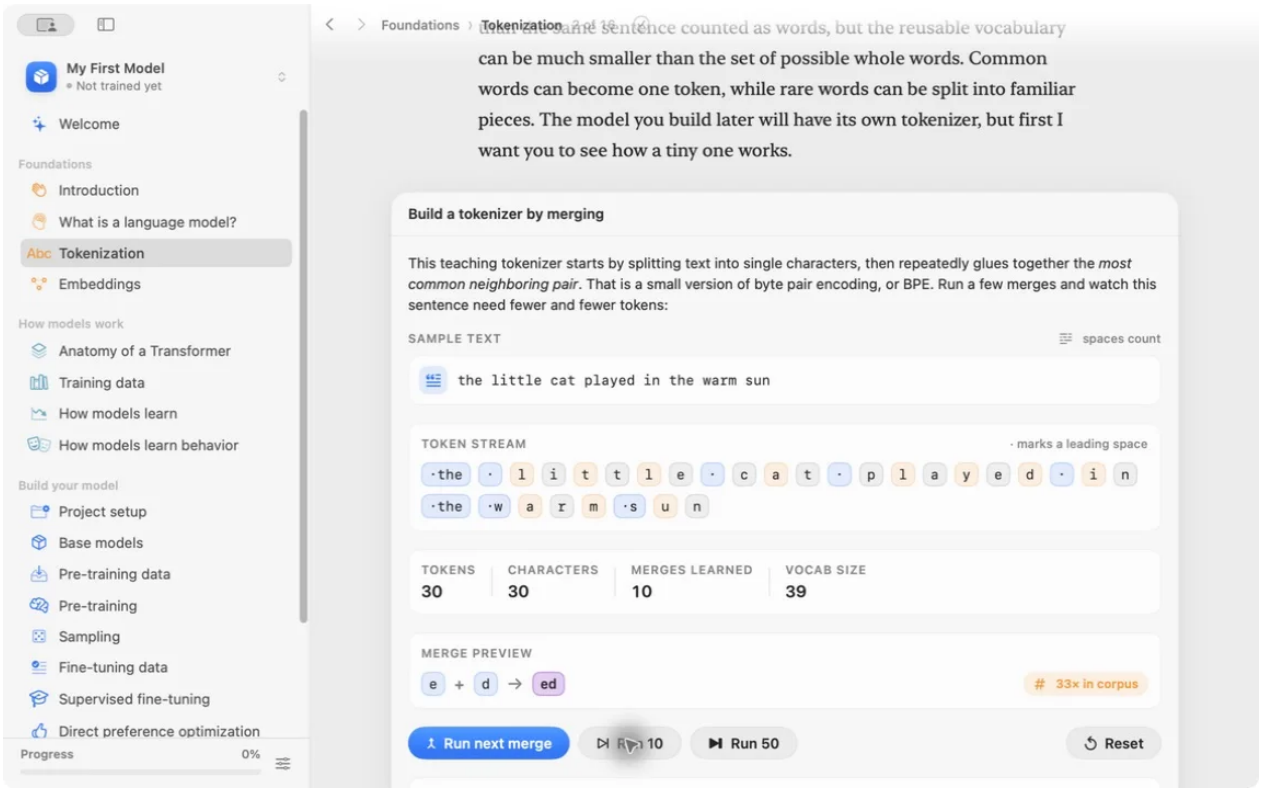
Sequence length affects compute. If a 60-character prompt becomes 20 tokens, a context length of 64 leaves room for 44 more input or generated tokens. A different tokenizer may split the same prompt into 35 tokens and leave less room.

What to observe: familiar patterns tend to use fewer reusable pieces. Rare text often splits more.

Run the tokenizer playground

Watch byte-pair encoding merge frequent neighboring pieces and connect vocabulary size to the sequences the model will process.

The tokenizer playground starts with small pieces, then repeatedly merges frequent neighboring pairs.



Reset the playground. Run one merge at a time for the first five merges. Record the pair, its frequency, and how the token count changes.

For a simplified corpus containing `low low lower`, the first representation might look like this:

```
l o w  l o w  l o w e r
```

If `l o` is the most frequent adjacent pair, merging it creates `lo`. A later merge can create `low`. The algorithm follows counts, not dictionary definitions.

Then run ten or fifty merges. Compare the final token stream with the first one.

A larger vocabulary can encode familiar text with fewer tokens. It also increases the embedding table and the number of output scores predicted at every position.

Measure both sides in your notebook:

```
compression = characters in sample / tokens in sample
embedding parameters = vocabulary size × embedding width
```

For example, growing from 2,000 to 4,000 tokens at width 256 adds roughly `2,000 × 256 = 512,000` embedding values. Weight tying and architecture details can change the exact total, but the tradeoff remains.

Run the same sample from the training domain and an out-of-domain sample. A tokenizer tuned to stories may compress code or another language poorly.

What to observe: the algorithm follows frequency. It does not understand that a piece is a noun or verb.

Explore embedding space

Use the embedding map to see learned proximity while remembering that the two-dimensional view is only a projection.

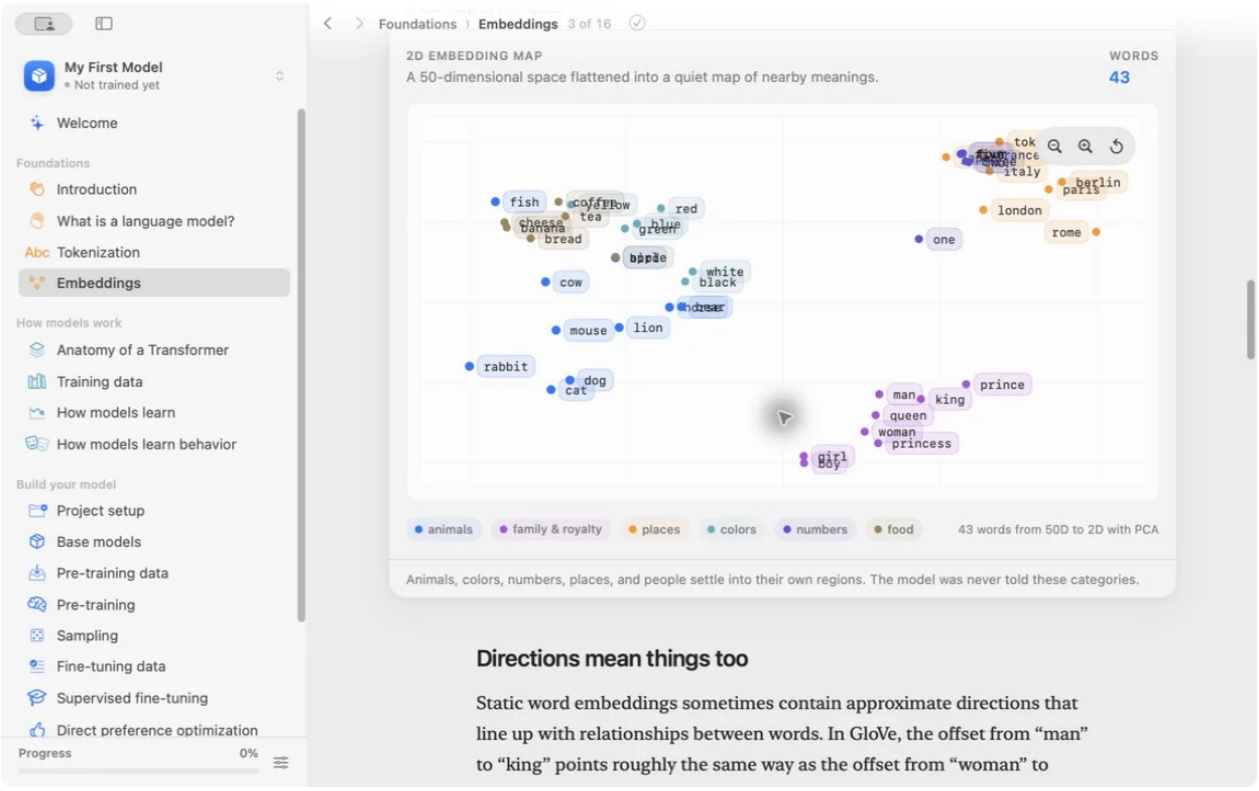
An **embedding** is a learned vector assigned to a token ID. It gives the transformer a numerical representation to update with context.

With embedding width four, one token might begin like this:

```
cat → [0.12, -0.40, 0.08, 0.31]
```

The values are learned during training. Nobody labels one dimension “animal” or another “grammar.”

Open **Embeddings** and inspect the map.



Pick one animal, one color, and one place. Find their nearest visible neighbors. Write down one cluster that makes sense and one relationship that looks weak.

The original vectors have many dimensions. The app flattens them into two dimensions so we can see them. Some distances and overlaps are distorted by that projection.

The map also shows one vector per token, not a fixed meaning for a word in every sentence. Contextual representations inside later transformer layers change with surrounding tokens.

Compare an early and late checkpoint if the tool allows it. Use the same tokens and projection settings. A cluster moving is an observation; claiming the model "understands animals" requires stronger evidence.

Nearest neighbors can reveal dataset artifacts too. Two names may be close because they repeatedly appeared in the same story template, not because they share a real-world category.

What to observe: related use can produce nearby vectors without anyone manually entering category labels.

Trace a transformer prediction

Follow one token through embeddings, attention, transformer blocks, and output scores without hiding the prediction behind a chat interface.

Open **Anatomy of a Transformer** in the app.

Start with a short tokenized prompt such as `[the, cat, sat]`. At every position, a decoder-only transformer predicts the following token.

Each token ID becomes an embedding and receives position information. Masked self-attention lets a position combine earlier context without looking at future target tokens. A feed-forward network transforms each position. Residual paths carry information around both operations.

At the end, the model produces one score for every token in the vocabulary. Softmax turns those scores into probabilities.

Suppose three candidate scores become these probabilities:

```
" on"   0.60
" near" 0.25
" blue" 0.15
```

Training compares this distribution with the actual next token. Generation chooses one candidate using the configured sampling rule, appends it, and runs the prediction path again.

Write one sentence in your notebook for each stage: input IDs, embeddings, attention, blocks, output scores, next-token choice.

Change one prompt token and inspect the next-token distribution again. The difference shows context affecting computation. It does not prove that one attention head alone caused the final choice.

The model stores parameters, not completed answers. A generated paragraph is many repeated conditional predictions.

What to observe: the model produces a distribution of possible next tokens, not one stored answer.

Check your understanding: model foundations

Check token IDs, BPE merges, embeddings, attention, transformer output, and the prediction error used for learning.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

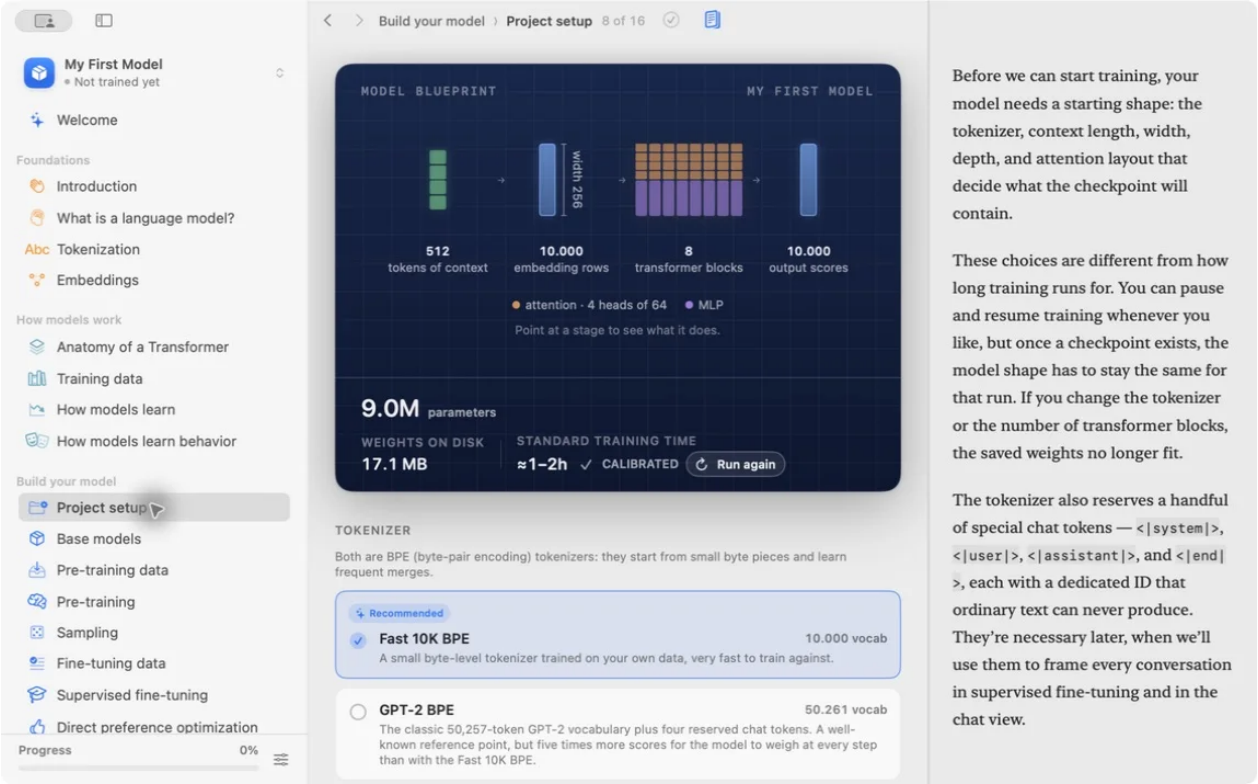
Design the run

Choose the model shape, tokenizer, data, and practical resource budget.

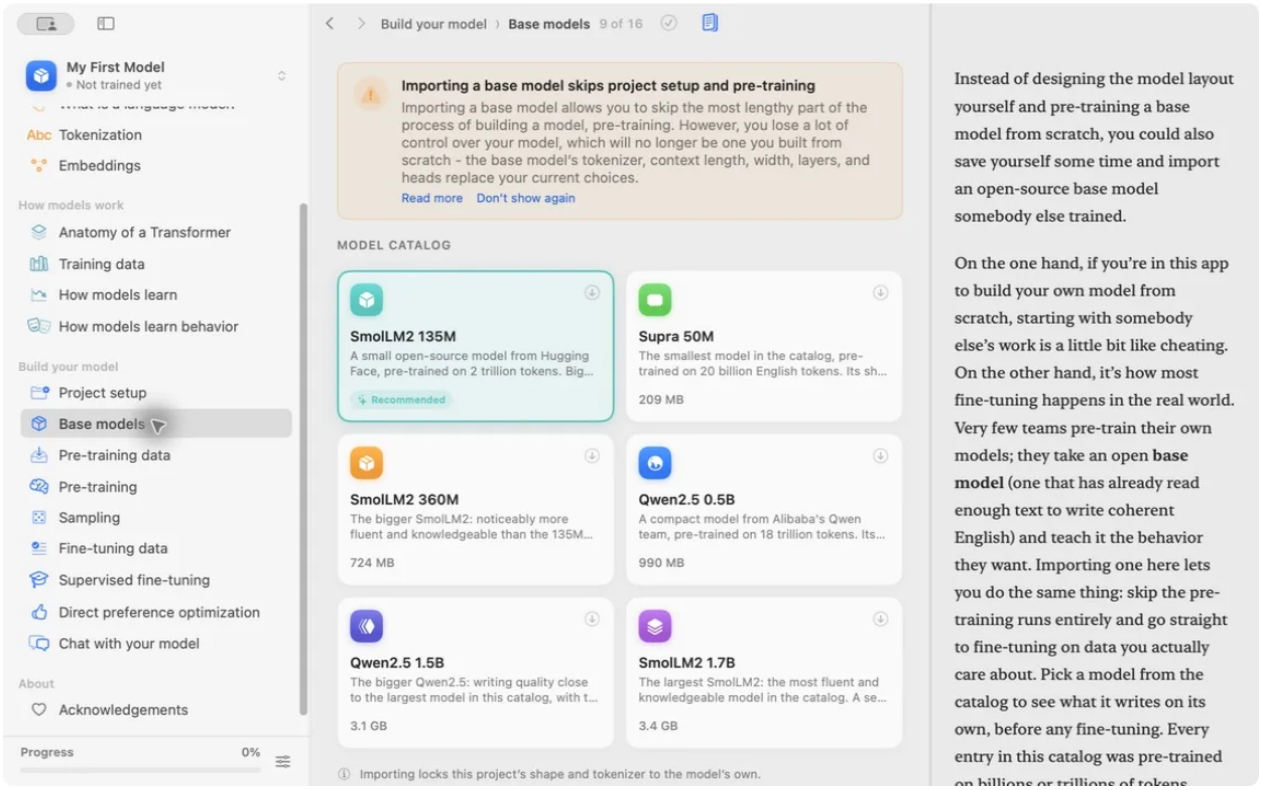
Choose a model blueprint

Read the model blueprint as a set of connected design choices and record the exact shape before training.

Open **Project setup**. Start with the app recommendation unless you have a specific comparison in mind.



If you start from an existing model instead, the current catalog recommends SmolLM2 135M. Importing it skips project setup and pre-training, and replaces the project's shape and tokenizer with the model's own. SFT, DPO, sampling, and chat remain available.



Record the vocabulary size, context length, embedding width, number of transformer blocks, and attention-head layout.

These values are connected:

- vocabulary size shapes the token embedding and output scores
- context length limits how many positions one example can contain
- embedding width controls the size of representations
- block count controls depth
- head count divides attention work across several learned projections

The embedding width must divide cleanly across the attention heads in common designs. With the Starter width of 256 and four heads, each head works with 64 features.

A rough decoder-transformer estimate is useful before trusting the displayed total:

```
token embeddings   vocabulary size × width
each block   a constant multiple of width²
```

The exact constant depends on feed-forward size, biases, normalization, and whether input and output embeddings share weights. Use the app's exact count for the run and the estimate as a sanity check.

These choices define tensor shapes inside the model. A checkpoint created for one shape cannot load into a different shape without an explicit conversion.

Give the run an identifier in your notebook, such as `starter-tinystories-20k`.

Do not tune five fields for the first run. Accept the recommended blueprint, record it, and preserve your limited compute for a clean baseline.

What to observe: changing one blueprint field changes parameter count, weight size, or the calibrated estimate.

Parameters, memory, and training estimates

Interpret the parameter count, weight size, training memory, and time estimate without treating any one number as the whole cost.

Parameters are the learned weights changed during training. More parameters give a model more capacity, but they also increase compute and memory use.

Estimate the weight storage first:

```
weight bytes  parameter count × bytes per stored parameter
```

A 9-million-parameter model stored with two bytes per value needs roughly 18 MB for weights. File metadata and tensor packaging add some overhead.

The saved weight file is not the full training-memory requirement. Training can also keep gradients, optimizer state, temporary buffers, and activations for every batch and layer.

Activations grow with batch size, sequence length, width, and depth. This is why halving context length or batch size can rescue a run even when the parameter count stays fixed.

Record the app's parameter count, weight size, and standard training estimate. Note your Mac chip and unified memory beside them.

Treat the time as an estimate. Thermal conditions, background activity, dataset work, and configuration can change throughput.

Turn throughput into a second estimate when the app reports tokens per second:

```
training time  total training tokens / tokens per second
```

If 10 million training tokens run at 2,000 tokens per second, the compute portion is about 5,000 seconds, or 83 minutes. Validation, checkpoint writes, startup, and changing throughput add time.

If the estimate is longer than you can leave the Mac available, reduce the run before starting. A completed small experiment is more useful than an abandoned oversized one.

Keep memory headroom for macOS and the application. A configuration that barely fits may slow dramatically or fail when another process uses unified memory.

Choose a tokenizer

Choose a tokenizer that fits the model and dataset, then record the vocabulary decision before checkpoints make it fixed.

For the first from-scratch run, use the app's recommended small BPE tokenizer. This keeps our baseline close to a known working configuration.

A larger vocabulary can shorten sequences for familiar text, but every extra vocabulary entry adds output scores the model must learn.

Preview the tokenizer on text from your chosen dataset. Look for excessive splitting of important names, symbols, or language-specific characters.

Use one fixed sample and record two measurements:

```
tokens per 1,000 characters
percentage of examples longer than the context limit
```

The first measures compression. The second reveals truncation pressure. A tokenizer can look

efficient on one paragraph while splitting important parts of the dataset poorly.

Fit or choose the tokenizer using training data only. If tokenizer construction sees validation and test text, those splits are no longer completely held out from the pipeline.

Record the tokenizer name, vocabulary size, and why you chose it.

If you import a base model, keep its tokenizer. The saved weights expect that exact vocabulary and ID mapping.

Save the tokenizer artifact beside the checkpoint. A vocabulary size alone cannot reconstruct which piece received each token ID.

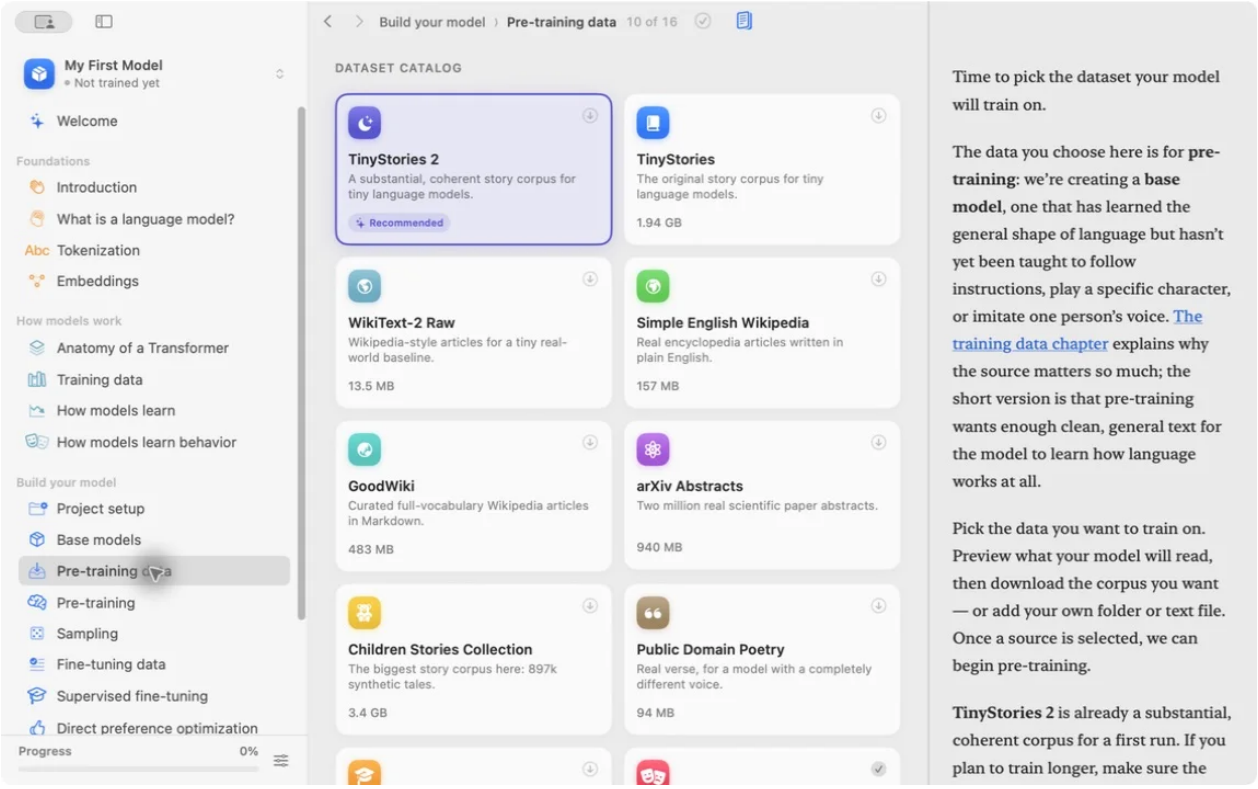
Run one round-trip test, one domain sample, and one out-of-domain sample. Keep the results in the notebook before training makes this choice expensive to revisit.

What to observe: tokenizer and model shape are part of one compatible system.

Select and prepare a dataset

Choose training data that fits the experiment, inspect its source and license, and preview the actual text before downloading or importing it.

Open **Pre-training data**.



For a first small run, the recommended story corpus gives the model a narrow and visible target. A tiny encyclopedia sample tests a different kind of text.

Read the source, license, size, and preview. Search the preview for broken encoding, repeated boilerplate, private information, or content you would not want the model to imitate.

If you import your own text, keep a copy of the exact input and note how you cleaned it. Do not include secrets, private conversations, or material you do not have the right to use.

Split documents before creating overlapping token windows. If neighboring windows from one story land in both training and validation, validation loss can look better because it contains near-duplicates of training text.

Use three roles:

- **training** examples update parameters
- **validation** examples guide checkpoint and run decisions
- **test** examples stay untouched until the final comparison

Deduplicate before splitting when possible. Keep the cleaning and split rules deterministic, with a seed or stable document IDs.

Count tokens after tokenization, not only files or characters. Two datasets with the same download size can produce very different training-token budgets.

Write the dataset name, version or retrieval date, license, size, and reason for choosing it in your notebook.

Add a compact data card:

```
Source and license:
Documents before/after cleaning:
Training/validation/test token counts:
Duplicate-removal rule:
Known gaps or unwanted content:
```

Read twenty random examples from each split. Aggregate statistics will not reveal broken markup or a repeated footer by themselves.

Check your understanding: design the run

Check checkpoint compatibility, parameter and memory costs, estimates, tokenizer choices, and responsible dataset selection.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

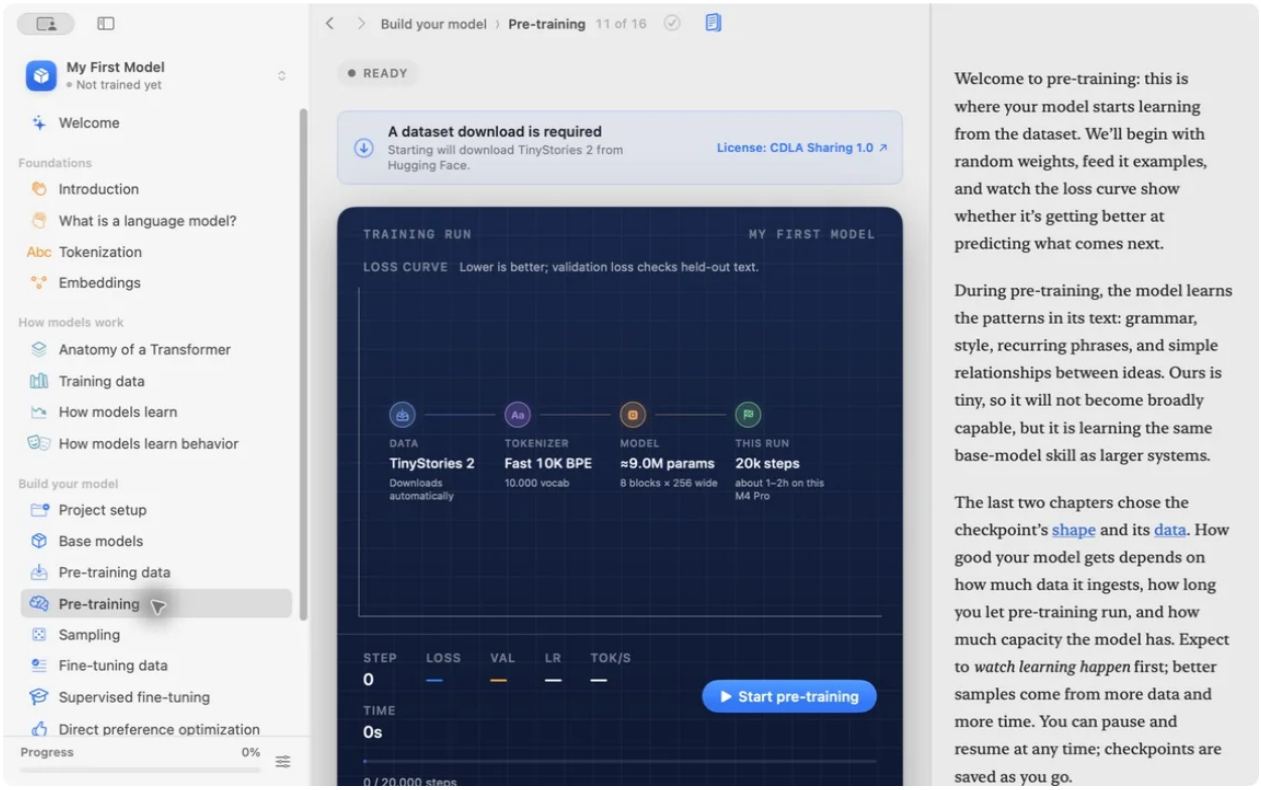
Pre-train and observe

Run training over time, read loss, sample checkpoints, and diagnose results.

Start a pre-training run

Review the complete run summary, start pre-training, and record the initial measurements before leaving the model to learn.

Open **Pre-training** and review the summary before pressing Start.



Confirm the dataset, tokenizer, parameter count, number of steps, and estimate match your notebook.

Pre-training creates shifted input and target sequences. For token IDs `[11, 24, 8, 5]`:

```
input:  [11, 24, 8]
target: [24, 8, 5]
```

The model predicts every target from the tokens to its left. Cross-entropy loss penalizes low probability on the actual next token.

A **step** processes one batch and updates the parameters once. It is not the same as one example or one complete pass through the dataset. Record batch size, sequence length, and step budget so the amount of training is understandable.

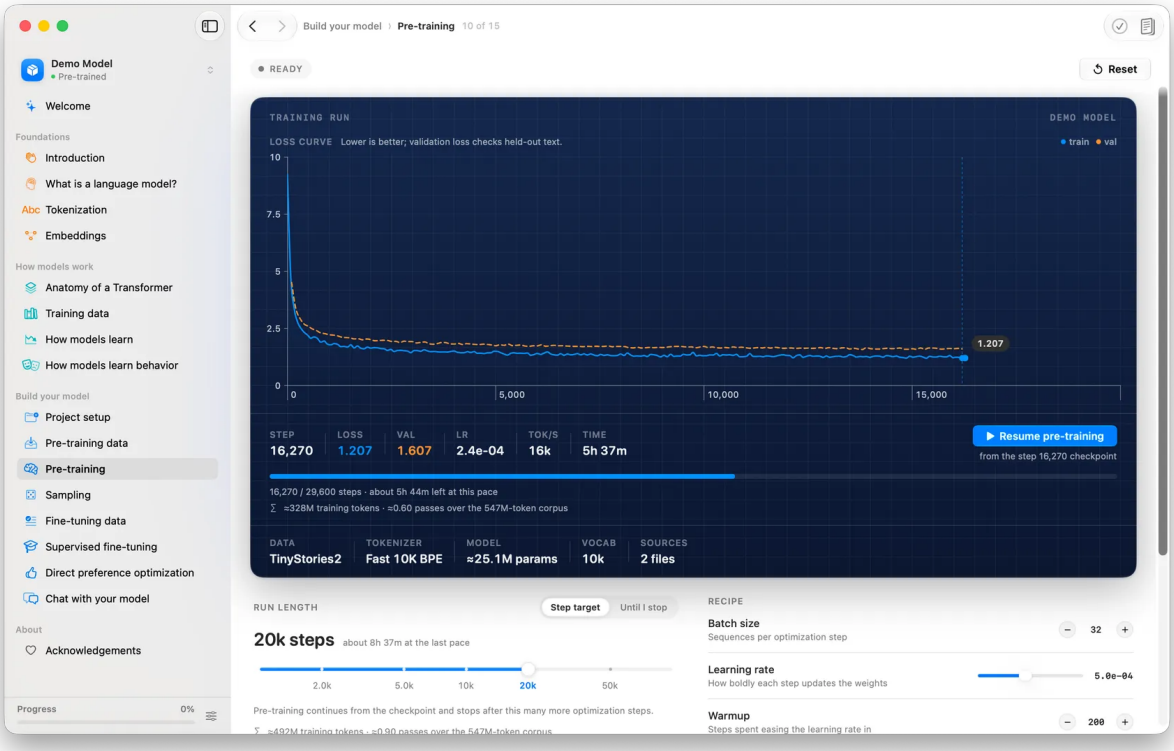
Start the run. Record the start time, first reported throughput, initial training loss, and first validation loss when available.

Do not judge text quality from the first sample. Randomly initialized weights need time before output resembles the dataset.

Check the first minutes for broken plumbing:

- loss is finite rather than `NaN`
- throughput is nonzero and reasonably stable
- validation uses the expected held-out split
- checkpoints and samples appear at the configured interval
- memory pressure does not make the system unusable

If the run fails immediately, preserve the configuration and error before retrying. Changing several settings destroys the evidence about which input caused the failure.



Pause and return later

Treat model training as long-running work by recording state, confirming a checkpoint, and resuming the course without waiting on one page.

Training can take hours or days. This course does not assume the run finishes while you read.

Let the run reach a saved checkpoint. Record the current step, loss values, elapsed time, and latest sample in your notebook.

A useful resume checkpoint needs more than model weights. Exact continuation commonly needs optimizer state, step and schedule position, and the data-order or random state used by the trainer.

If only weights are restored, training can still continue, but it is a new run segment rather than a bit-for-bit continuation. Record that distinction.

Pause or close the app using its normal controls. Reopen the same project and confirm the checkpoint history is still present before resuming.

After resume, compare these values with the notebook:

```
Run ID and checkpoint step
Optimizer and learning-rate state
Training and validation loss
Next checkpoint number
Sampling seed and baseline prompt
```

Loss may move slightly when a new batch arrives. A reset step counter, missing checkpoint, or large unexplained discontinuity needs investigation.

You can stop this course here. When you return, begin by checking that the project name, current step, and latest checkpoint match your notes.

If the run did not save correctly, preserve the error and logs before changing the project. A failed resume is useful diagnostic evidence.

Keep enough free disk for the next checkpoint. A training pause caused by a full disk can leave the newest artifact incomplete.

Read training and validation loss

Use both loss curves to distinguish learning, noise, a plateau, and a growing generalization gap.

Training loss measures next-token error on batches used to update the model. **Validation loss** checks held-out text that does not update the weights.

For one target token, cross-entropy is the negative logarithm of the probability assigned to that token:

```
target probability 0.50 → loss 0.69
target probability 0.10 → loss 2.30
```

The displayed loss averages many token predictions. Lower means the model assigned more probability to held-out targets, not that every generated sentence is better.

Look at the direction over many measurements, not one spike. Early curves can be noisy.

If both losses fall, the model is learning patterns that transfer to held-out data. If training loss falls while validation loss rises for a sustained period, the model may be overfitting.

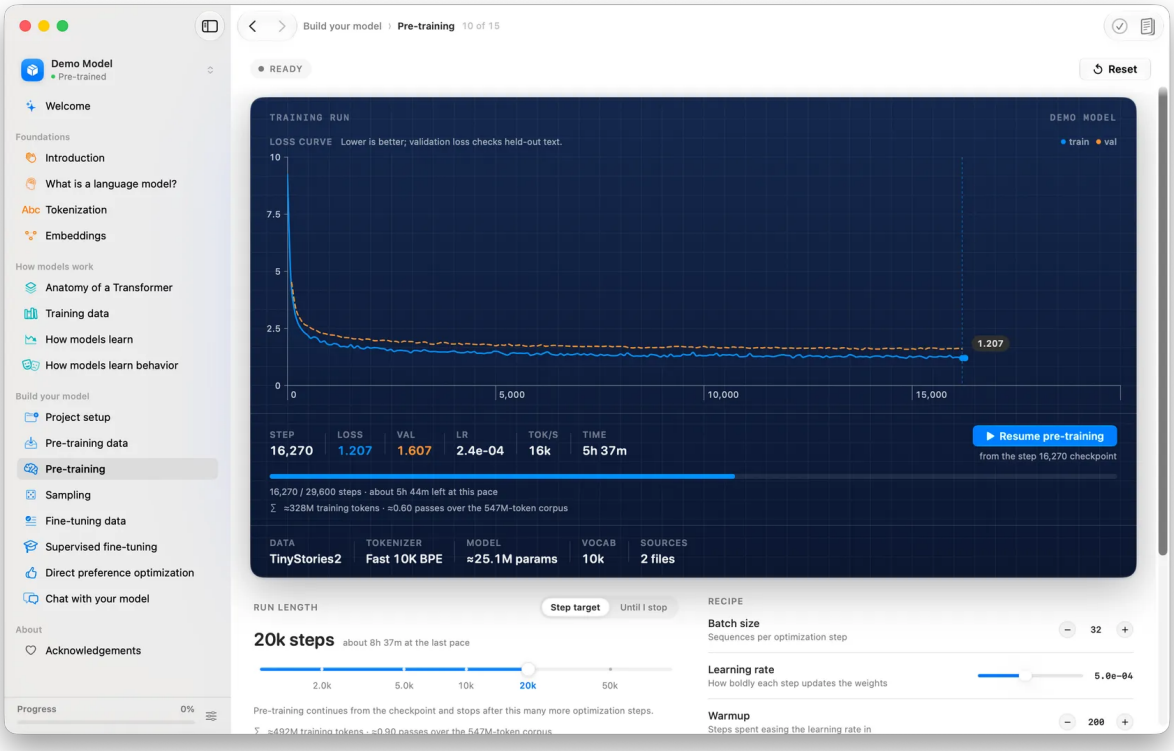
Record the step and value at the beginning, middle, and end of the run. Note the lowest validation loss and where it occurred.

The gap is evidence, not a verdict. A sustained falling training loss with rising validation loss suggests the model is fitting patterns that do not transfer to the validation split. A flat pair can mean the run plateaued, the learning rate is ineffective, capacity is limited, or data is too noisy.

Perplexity is sometimes reported as `exp(loss)`. A loss of `2` corresponds to about `7.4`, but compare it only under the same tokenizer and evaluation construction.

Do not compare raw loss values between unrelated tokenizers as if they were the same experiment.

Mark checkpoints on the curve and attach fixed-prompt samples. Loss evaluates target probabilities under teacher-provided context; free-running generation feeds the model's own sampled tokens back into later predictions.



Checkpoints and sampling

Compare saved model states with fixed prompts and sampling controls instead of choosing isolated outputs after the fact.

A **checkpoint** is a saved model state at one training step. It lets you resume and compare progress.

Choose an early, middle, and later checkpoint. Run the three baseline prompts against each one.

Keep temperature, top-k, top-p, min-p, maximum tokens, and random seed fixed where the app exposes them.

These controls act on one next-token distribution:

- temperature changes how sharp or flat the distribution is
- top-k keeps at most the highest-scoring `k` candidates
- top-p keeps the smallest high-probability set reaching a probability mass
- min-p removes candidates too weak relative to the best candidate
- the random seed controls reproducible draws when supported

Do not change every filter at once. For the baseline, choose one documented preset and keep it fixed.

Run at least two sampling conditions for a useful contrast:

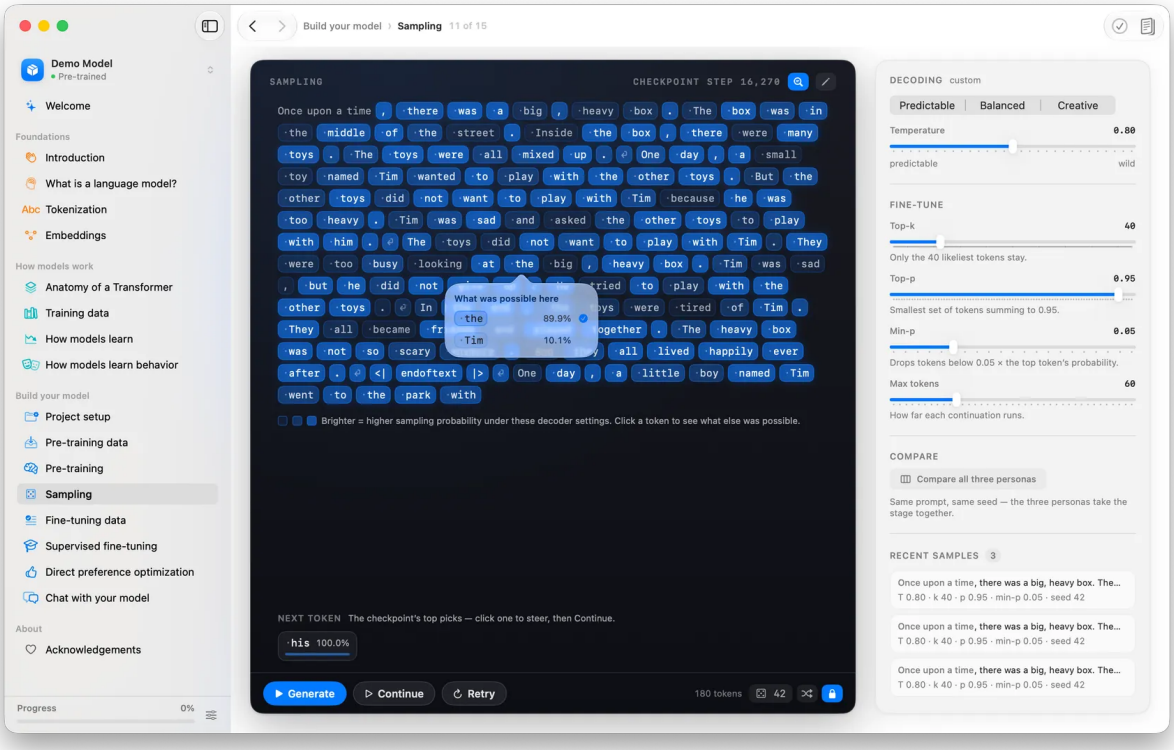
1. deterministic or very low-randomness output for checkpoint comparison
2. one moderate-randomness preset with three fixed seeds

One lucky stochastic sample is not evidence that a checkpoint improved.

Save every output, including weak ones. Add the checkpoint and settings above each sample in your notebook.

Look for changes in spelling, sentence closure, repetition, topic continuity, and similarity to the dataset.

If a later checkpoint produces worse text, check validation loss and several seeds before deciding it regressed. The latest weights are not automatically the best weights.



Diagnose poor results

Use the data, loss curves, samples, and configuration to choose one justified next experiment instead of changing everything together.

Start with the symptom. Is the output still noise, repetitive, copied from data, off-topic, or grammatically plausible but incoherent?

Then check the evidence:

- If loss never improves, check data loading, tokenization, and the run configuration.
- If training improves but validation worsens, stop near the best validation checkpoint or change capacity and data.
- If loss improves but samples remain weak, compare prompts and sampling settings before retraining.
- If output copies long passages, inspect duplication and train/validation separation.

Add two checks before spending more compute:

- Decode a batch of training tokens. If the text is broken, training longer will not repair the input pipeline.
- Ask the model to overfit a tiny batch in a disposable run. If loss cannot fall there, suspect the objective, shapes, learning rate, or optimizer path.

Use this evidence table:

Symptom	First measurement	One possible next test
Loss is `NaN`	First bad step and batch	Lower learning rate or inspect invalid values
Both losses stay flat	Gradient and learning-rate behavior	Tiny-batch overfit test
Validation rises	Best validation checkpoint	Stop earlier with the same run
Repetition only in samples	Token probability trace	Change one sampling control
Memorized passages	Nearest training matches	Deduplicate and rebuild the split

Choose one main change: cleaner data, more appropriate data, a different run length, or a revised blueprint. Give the new run a new identifier.

Keep the failed run. It is the comparison that explains why the next decision exists.

Match the fix to the bottleneck. More parameters can worsen overfitting. More data can waste compute when tokenization is broken. More steps can amplify memorization after validation stopped improving.

State the expected result before running the change. “Lower the learning rate to prevent the early loss spike” is testable. “Try different settings” is not.

Check your understanding: pre-train and observe

Check held-out validation, overfitting, checkpoints, comparable sampling, long-running work, and controlled diagnosis.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Shape model behavior

Use SFT, preference data, chat, and inspection tools deliberately.

Run supervised fine-tuning

Teach a pre-trained model a prompt-and-response pattern, then compare its behavior with the same baseline prompts used before SFT.

Pre-training learns to continue its corpus. **Supervised fine-tuning**, or SFT, trains on examples of the interaction pattern we want.

Open **Fine-tuning data**. Preview the examples before selecting a dataset. Check which fields represent the user and assistant turns.

The formatting is part of the data. A conversation might become one token sequence:

```
<user> Summarize the note.
<assistant> The release is ready.
```

Many SFT setups calculate loss mainly or only on the desired assistant response. Others train on more of the formatted sequence. Record what the app does because it changes the objective.

Choose a small dataset whose response style you can recognize. Record its source, license, size, and one example.

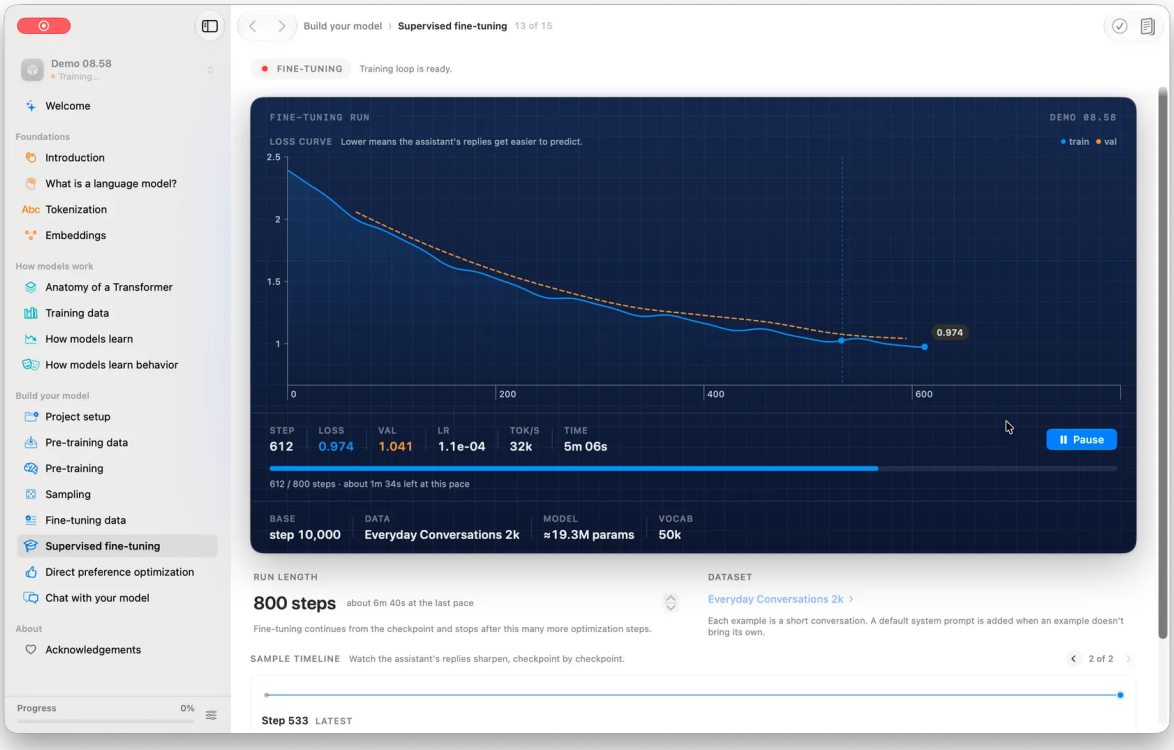
Reject examples with contradictory instructions, missing responses, hidden private data, or formatting that does not match the chat template used at evaluation time.

Run SFT from the checkpoint with the best validation result, unless your notebook records a different reason.

After it saves a checkpoint, repeat the baseline prompts with the same sampling settings. Record what changed and what did not.

Add three in-domain instruction prompts and three held-out variations. A model can imitate the exact training format without learning to generalize the behavior.

Compare pre-training capabilities too. SFT can improve instruction following while narrowing style, increasing refusals, or weakening ordinary continuation. Keep the base checkpoint available.



Preference data and DPO

Use preferred and rejected response pairs as a separate behavior experiment after establishing a clear SFT baseline.

Direct preference optimization, or DPO, learns from comparisons.

One example contains a prompt, a preferred response, and a rejected response. The difference should represent a behavior you genuinely want.

For example:

```
Prompt: Give one release risk.
Preferred: The update path has not been tested from version 1.0.
Rejected: Everything looks great! Here are five unrelated ideas...
```

This pair expresses focus and evidence. A pair where the preferred response is merely longer teaches

an accidental length preference.

Open **Direct preference optimization** after creating an SFT checkpoint. The preference ballot asks the same prompt twice. Choose the answer you would rather receive, and the app saves the prompt, chosen answer, and rejected answer.

If both answers are identical, the pair is not saved. Sample again. Reject ambiguous pairs where both answers are equally good or where the preferred answer is only longer.

Four saved pairs unlock DPO. That is enough to see the machinery work, not enough to define a broad preference. Apply the same judgment consistently and collect more varied pairs when you want a clearer signal.

The app holds some pairs back to measure preference accuracy. With a tiny set, that number can jump sharply. Record the training and held-out pair counts instead of treating one percentage as stable.

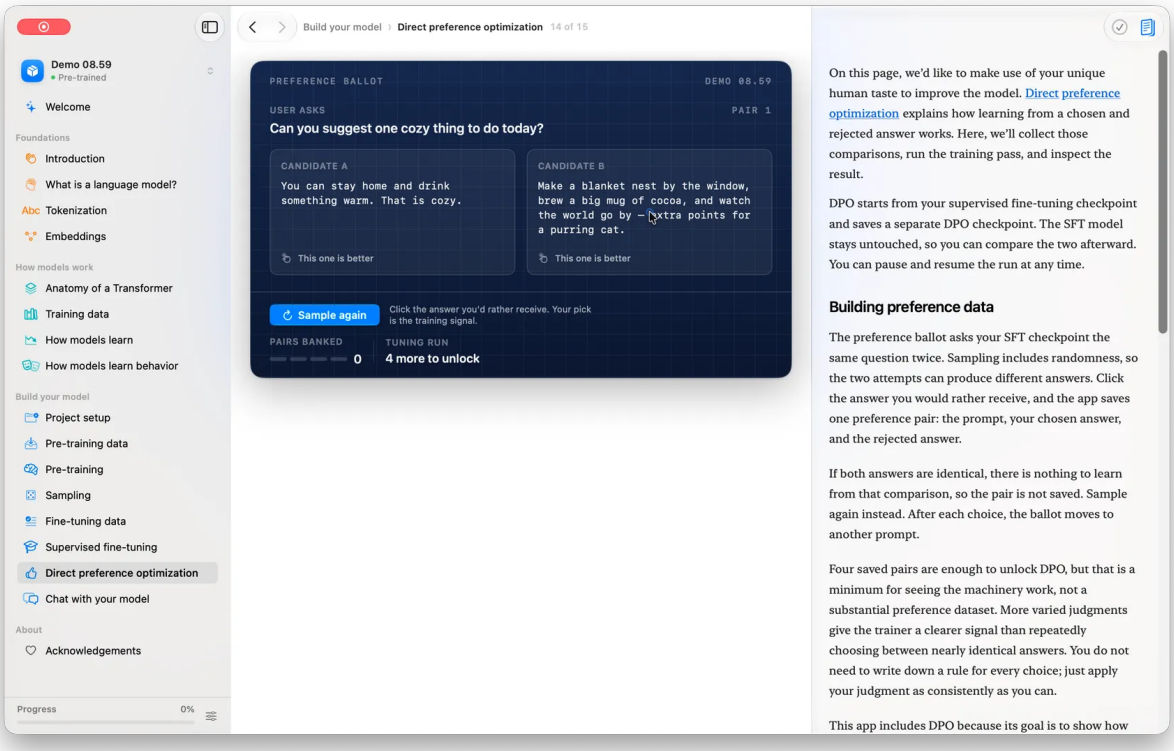
Run a small DPO experiment after SFT. Repeat the same evaluation prompts and settings.

DPO adjusts the trained policy relative to the frozen SFT checkpoint. The two models start with identical weights, so the app's logistic DPO loss begins near `ln(2)`, or `0.693`. Record the pair count and training steps rather than treating every DPO run as equivalent.

Record both improvements and regressions. Preference optimization can move behavior in an unwanted direction when the pairs are weak or narrow.

Check for reward-like shortcuts: excessive brevity, repeated disclaimers, copied phrases, or a style that wins the training pairs but fails new prompts.

The clean comparison is `SFT checkpoint → DPO checkpoint`, with the same chat template, prompts, seeds, and sampler. Do not attribute a difference to DPO when the generation settings also changed.



Chat with your model

Use chat as a repeatable evaluation interface by keeping prompts, checkpoints, and sampling settings attached to every observation.

Open **Chat with your model** and choose the checkpoint you want to test.

Begin with the baseline prompts. Then add three instruction-shaped prompts that match the SFT data and three that do not.

Chat adds formatting around every turn. The current app lets you edit or disable the system instruction. Record its exact text with each comparison. A base model prompted as plain continuation and an SFT model prompted through a chat template are not receiving the same token sequence.

Switching between the base, SFT, and DPO checkpoints starts a fresh chat. Save the transcript before switching, then reuse the same first prompt and sampling settings.

For every result, record the checkpoint and sampling settings. Do not compare one low-temperature answer with another high-temperature answer without saying so.

Use a small evaluation matrix:

Prompt type	Base	SFT	DPO
Seen instruction shape			
Held-out instruction shape			
Story continuation			
Out-of-domain question			
Malformed request			

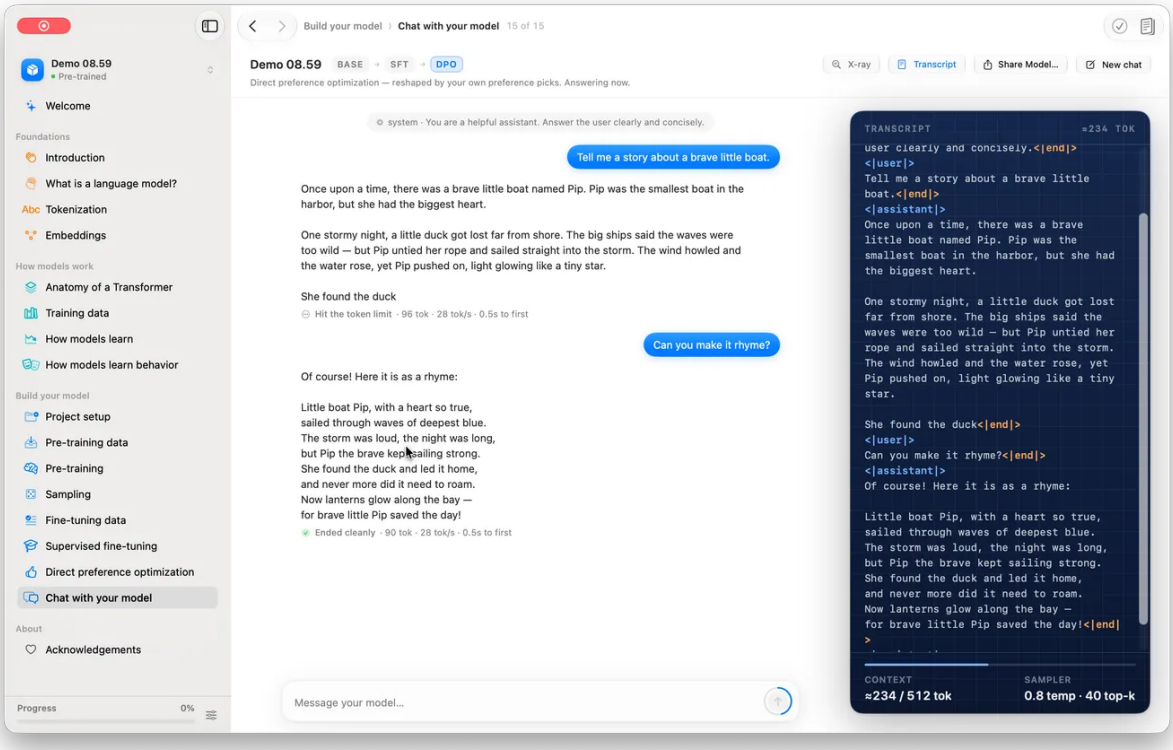
Run multiple fixed seeds for stochastic sampling. Record complete transcripts because previous turns consume context and influence later tokens.

Watch the context counter. Once old turns are truncated or the context limit is reached, a later answer is not directly comparable with a fresh one-turn prompt.

Test one malformed request and one request outside the dataset's domain. A useful report includes failure boundaries.

The chat view makes the model feel familiar, but the underlying system still predicts one token at a time.

Do not grade fluency as truth. A small model can produce a smooth unsupported claim. Score instruction following, format, repetition, and factual support separately.

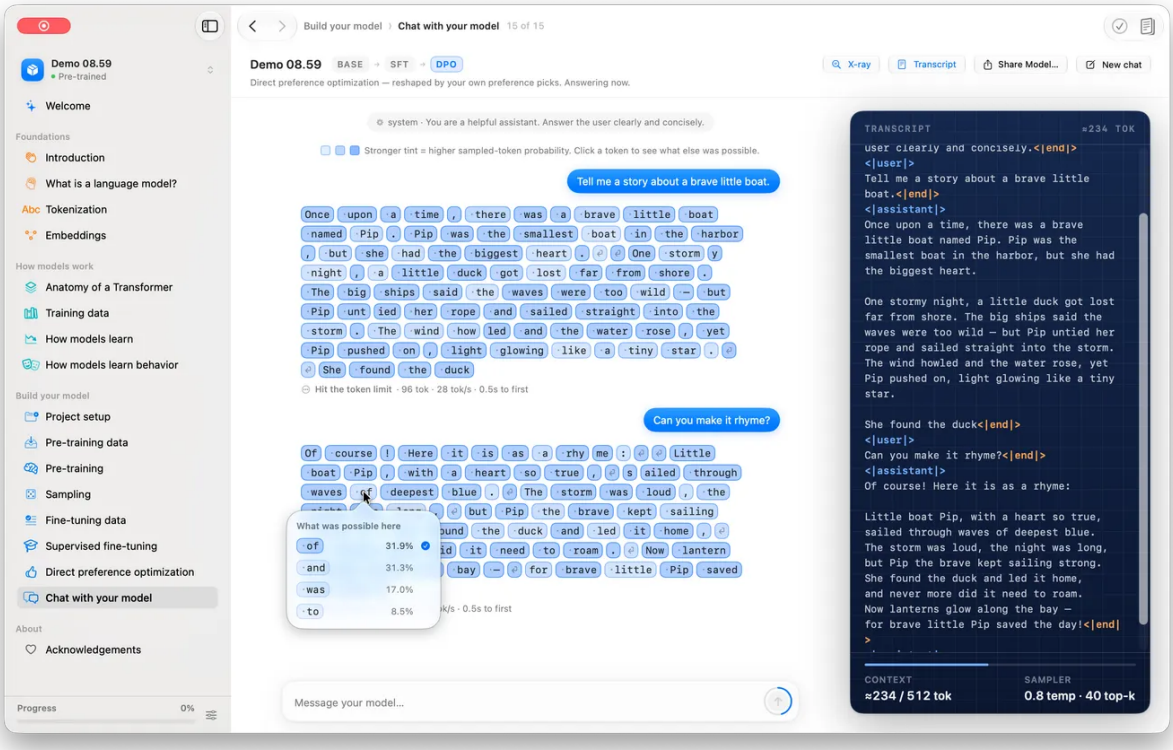


Inspect the model with visualization tools

Use token probabilities, alternatives, embeddings, and attention views to turn a surprising output into a concrete question.

Choose one generated answer that surprised you. Open the app's X-ray or inspection view for that generation.

Find a token where the model had several plausible alternatives. Record the chosen token, its probability, and the next two alternatives. The X-ray reports probabilities after the active temperature and filters, so save those settings too.



Ask a counterfactual question: what changes if one prompt token changes? Run both prompts with the same checkpoint and sampling controls, then compare the distributions at the relevant position.

Look back at the prompt and ask which context tokens might have influenced that choice. Use the attention view when available, but do not treat one bright attention weight as a complete explanation of the model.

Attention weights show how one operation combined positions. They do not include every residual path, feed-forward transformation, or later layer that contributed to the final logits.

Use the embedding map to inspect one important token from the prompt. Remember that its two-dimensional position is a projection.

Write a narrow observation: "At this position, the model gave these alternatives similar probability." Do not claim the visualization reveals a human-like thought process.

Use visualizations to propose tests, not finish explanations. If you suspect the model relies on a name in the prompt, remove or replace that name and measure the output distribution again.

Save the checkpoint, exact prompt tokens, layer or head selection, and visualization settings. A screenshot without those details is difficult to reproduce.

Check your understanding: shape behavior

Check SFT examples, preference pairs, controlled chat comparisons, token probabilities, and when to add DPO.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Compare and report

Compare checkpoints fairly and document what the experiment proved.

Compare checkpoints fairly

Build one comparison table that holds prompts and sampling settings constant across base, pre-trained, SFT, and DPO checkpoints.

Create a table in your notebook with one row per prompt and one column per checkpoint.

Use the same baseline prompts, generation length, temperature, top-k, top-p, min-p, and seed where available.

Compare model stages separately:

```
random initialization → pre-trained checkpoints
best pre-trained checkpoint → SFT checkpoint
SFT checkpoint → DPO checkpoint
```

Each transition answers a different question. Combining them into "the final model is better" hides which training stage changed the behavior.

Score only things you can explain. Useful columns include complete sentence, repetition, follows requested format, stays on topic, and unsupported claims.

Define the rubric before reading final samples. For a binary item such as "follows requested JSON format," write the exact passing condition.

Use several fixed seeds when sampling is stochastic. Report a count such as `4 of 5 outputs completed the sentence`, not only the best example.

Keep the raw samples under the table. A short score without its output is hard to audit.

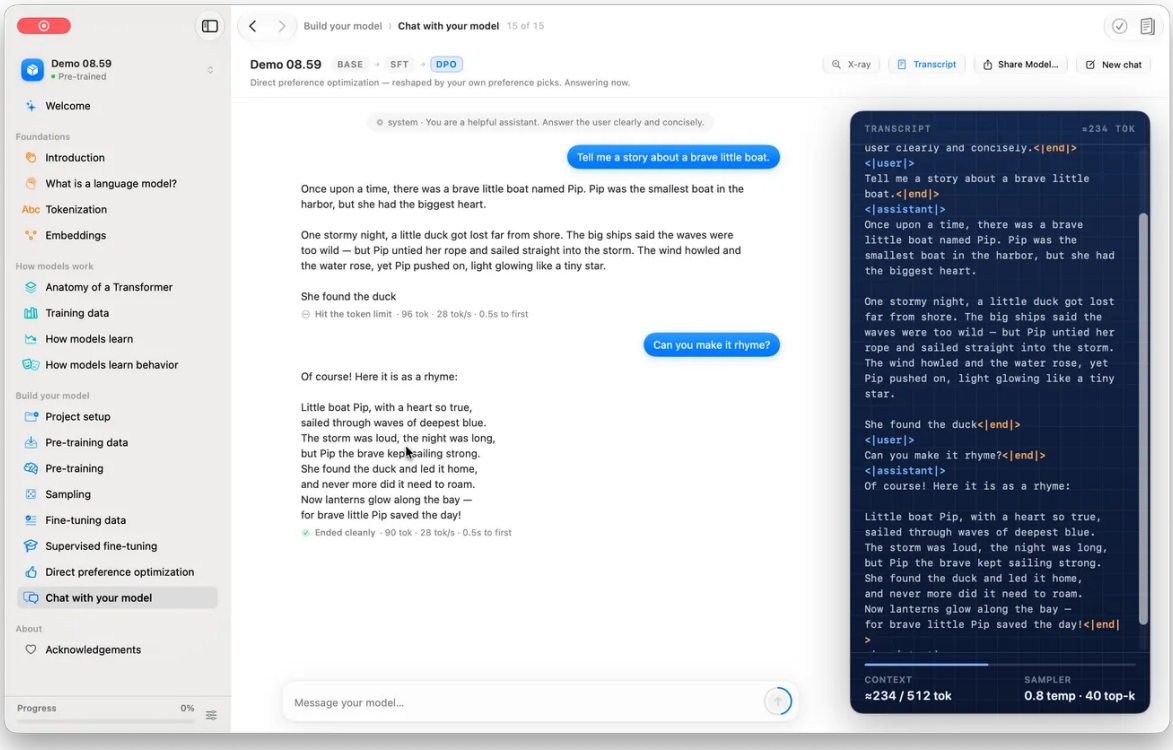
Add loss and compute context beside behavioral results:

Checkpoint	Training step	Validation loss	Training time	Notes
Early				
Best validation				
Final				

Check evaluation prompts against the training data when possible. A copied answer is evidence of memorization, not held-out generalization.

Choose the checkpoint you would keep and write one evidence-based reason. The latest checkpoint does not have to be the best one.

If two checkpoints are close, say the experiment cannot distinguish them. Honest uncertainty is more useful than choosing a winner from noise.



Write the model lab report

Finish a practical report that records configuration, data, loss, samples, fine-tuning results, and limitations without requiring submission or grading.

Copy this outline into `model-lab.md` and complete every section:

```
# Model lab report

## Goal
## Mac and app version
## Run manifest
## Model configuration
## Dataset and tokenizer
## Data preparation and split
## Parameter count and estimates
## Compute used
## Training observations
## Training and validation loss
## Checkpoint samples
## Evaluation method
## SFT results
## DPO results
## Final comparison
## Limitations
## Next experiment
```

Include exact configuration values, dataset source and license, parameter count, run duration, important loss points, and unedited sample outputs.

Connect every conclusion to evidence:

```
Claim: the middle checkpoint generalizes better than the final checkpoint.
Evidence: lower validation loss and higher fixed-prompt rubric scores across five seeds.
Limit: the test set contains only short story prompts.
```

Report compute and data honestly. Include processed training tokens, step count, checkpoint interval, elapsed time, and approximate energy or hardware details if you measured them. Do not convert an estimate into a measured value.

If you skipped DPO, say so and explain why. A skipped optional stage is better than an invented result.

End with one next experiment that changes a single variable. Example: keep the blueprint and tokenizer, then compare two datasets with the same step budget.

State what stays fixed and what result would support the hypothesis:

```
Change: replace the story dataset with the encyclopedia sample.
Fixed: tokenizer, blueprint, token budget, seed, and evaluation settings.
Expected evidence: lower held-out loss on encyclopedia text, weaker story-style scores.
```

Read the report once and remove any claim the recorded evidence does not support.

Attach artifacts or stable identifiers for the tokenizer, data manifest, checkpoints, raw outputs, and app version. The report should let a future run reproduce the comparison without relying on memory.

Final check: compare and report

Check fair checkpoint comparisons, configuration records, missing evidence, honest limitations, and conclusions supported by one experiment.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/language-model-builder/.