Learning Modern AI by Building It on Apple Silicon

AI From Tensors to Agents on Mac Silicon

Hassan Uriostegui

Waken AI Labs

AI From Tensors to Agents on Mac Silicon

Learning Modern AI by Building It on Apple Silicon

Hassan Uriostegui

Published by Lulu.com

Editorial brand: Waken AI Labs

First edition · 2026

ISBN 978-0-557-95 054-6

The ISBN barcode appears only on the print cover, in the Lulu-provided barcode area. Lulu.com is the ISBN imprint; Waken AI Labs is the editorial brand.

To my friend Arturo Castelan—a maestro in the wisdom and philosophy of life, whose guidance has guided others for more than two decades—and to my wife, Fernanda Beltran, who lights my days with her unstoppable strength and never gives up on me.

With special thanks to Brett O'Brien for his continued support and for opening the door to Silicon Valley. I'm grateful for the years we spent reimagining mobile social video at Viddy through short-form video and visual effects on mobile devices.

And to Zeus—Pakito, as we lovingly called him—the faithful companion behind so many long nights spent creating my projects. You were the essence of friendship in this world, my best friend. Your spirit is immortalized on this cover. Until we meet again in the infinite.

Contents

APPENDICES

About the Author

Hassan Uriostegui is a Mexican-American technologist and author whose work spans mobile video, visual effects, AI-human interaction, and AI research. He builds practical systems at the meeting point of creative technology and machine intelligence, and his work with Waken AI Labs explores responsible human-centered AI. This book turns that systems perspective into a reproducible learning path for Apple Silicon developers.

Acknowledgements

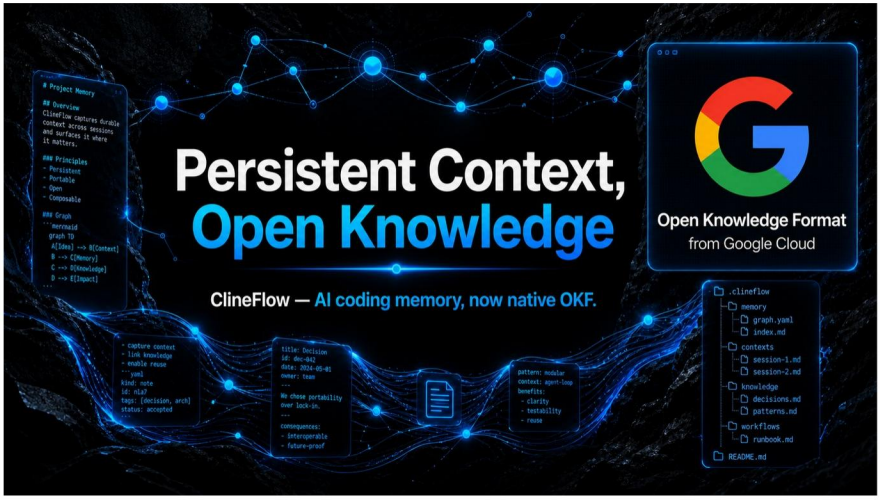
The companion repository is published as open-source software under the MIT License. The book text, diagrams, illustrations, and cover artwork remain all-rights-reserved as stated on the copyright page.

The Author's Toolkit

AI can shorten the distance between an idea and a working change. It does not, by itself, preserve why a decision was made, which command established a fact, or what a collaborator should do next. Those details are the context that makes an AI-assisted project resumable rather than merely fast for one session.

This book uses a small, optional tool from its author for that job: ClineFlow. ClineFlow is not required to read this book, run its experiments, or use the Book Intelligence Assistant. It is a development-memory layer for readers who want their preferred coding agent to work from a durable project record.

Optional reader tool: ClineFlow



ClineFlow and the Open Knowledge Format: persistent context, portable knowledge, and agent-readable project memory.

Repository: github.com/hassanvfx/clineflow

Read its current README first. If its workflow fits your project, install it with the documented command:

```
curl -fsSL https://raw.githubusercontent.com/hassanvfx/clineflow/main/install.sh | bash
```

The printed edition includes a QR code for this same repository link.

The problem: useful work disappears between prompts

A code change is only one part of engineering. A useful record also answers: what problem was being solved, what evidence was inspected, which assumptions were accepted, what failed, and what remains. Without that record, each new conversation begins by reconstructing context from scattered files and memory.

ClineFlow turns that reconstruction into a lightweight habit. Its documented workflow puts agent instructions beside the project, records work in journals, and treats a commit as a boundary where code and its decision record travel together. The result is readable without an agent: a collaborator can inspect the same Markdown and Git history that the agent receives.

A portable knowledge format, not another platform

ClineFlow documents a native Open Knowledge Format (OKF) bundle: Markdown documents with YAML front matter, indexes, links, and a dated log. The important idea is portability. Google Cloud describes OKF as an open specification for knowledge represented as ordinary Markdown files and YAML front matter, designed to be readable by people, parseable by agents, versioned with code, and independent of a particular vendor or runtime (Google Cloud 2026).

That does **not** mean that ClineFlow is Google-certified, Google-endorsed, or required by Google Cloud. It means that its documented approach aligns with the same practical premise: durable project context should be plain files that can move between tools instead of being trapped in a single assistant or service.

What the optional workflow provides

The current ClineFlow project documents five building blocks (Uriostegui 2026a):

Agent-facing instructions for tools such as Cline, Cursor, GitHub Copilot, and Windsurf.

A journal workflow for task history, decisions, validation, failures, and the next smallest action.

Knowledge indexes and logs that make the context navigable over time.

Optional local references to other repositories, kept outside Git so each developer can choose their own paths.

A dependency-light validator for the knowledge structure, with stricter parsing available when a project chooses it.

The core loop is intentionally modest: start a task, read the current context, record the decision and evidence as work progresses, then commit the code and record together. It is useful whether the work is a one-file fix, a research prototype, or a larger system with multiple repositories.

Review it before adopting it

Review the installed files before accepting them into an established project. Do not overwrite project-specific agent instructions, CI files, or repository rules blindly. ClineFlow is meant to add durable context, not to replace the engineering practices that already protect a codebase.

When to use it—and when not to

Use a journaled workflow when a project will span multiple sessions, when more than one person or agent needs the rationale behind a change, or when experiments must remain tied to the observations they support. It is especially useful when you regularly pause work and need a reliable way to resume.

Do not use the workflow as a substitute for tests, code review, security review, or explicit approval. A clean journal can explain a bad decision; it cannot make the decision correct. Keep secrets, private customer data, generated indexes, and machine-specific paths out of committed knowledge files.

A five-minute exercise

Choose one small change you expect to make today. Create a task journal using the template supplied by ClineFlow, then write four short entries: the intended outcome, one constraint, the command you ran, and the next action. Make the change, run its verification, and commit the journal with the code.

On your next session, begin by reading that entry before asking an agent to make another change. The exercise is successful if you can explain the state of the work without reconstructing it from chat history.

Takeaway

AI assistance becomes more useful when its context is durable, inspectable, and owned by the project. ClineFlow is one optional way to establish that practice; the rest of this book remains usable with ordinary Git, Markdown, and the experiments in this repository.

Optional: install ClineFlow

QR code for ClineFlow

ClineFlow: https://github.com/hassanvfx/clineflow

```
curl -fsSL https://raw.githubusercontent.com/hassanvfx/clineflow/main/install.sh | bash
```

ClineFlow is optional. Scan the code to read its current README on your phone, then use Safari Share → AirDrop to send the link to your Mac before installing. Keep the journal, agent instructions, and project-specific rules you already use; review the added files before committing them.

AI From Tensors to Agents on Mac Silicon

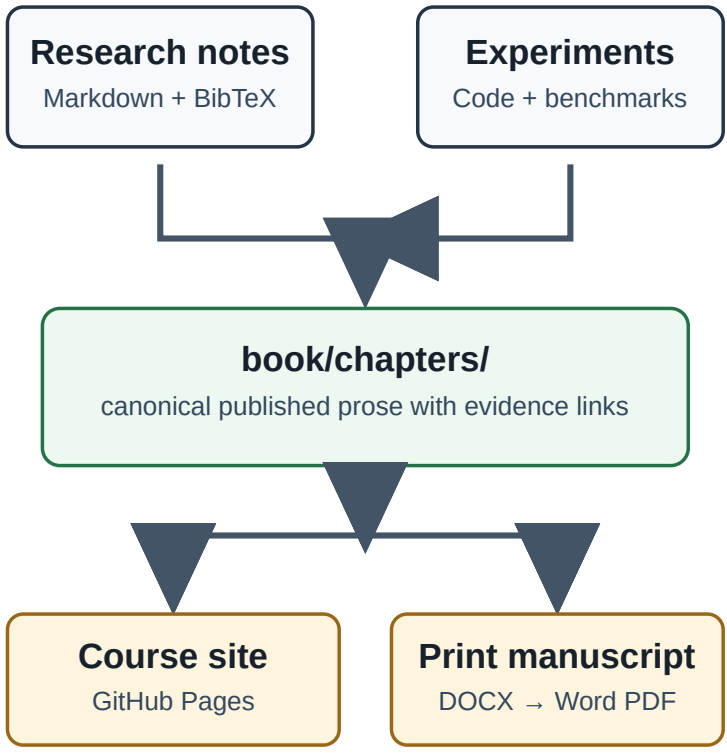
This book is a build log for a software engineer learning modern AI on Apple Silicon. Each chapter connects an idea to runnable code, an experiment, a failure mode, and a decision about when the technique is worth using.

This is also a living setup chapter. It records the exact way to prepare the project, run what is currently implemented, and distinguish a working lesson from a planned one. We will extend it as later milestones add transformers, local models, retrieval, and agent workflows; do not assume a command exists until it appears here or in the chapter that introduces it.

What you are building

The end product has one canonical editorial source and two reading formats:

One manuscript, two reading formats



Research notes and experiments inform canonical chapters, which produce the course site and print manuscript.

book/chapters/ is the only source of published prose. The course does not copy it.

research/ holds working notes and the shared references.bib bibliography.

experiments/ holds small runnable lessons; src/ contains reusable code.

benchmarks/ holds reproducible observations and their limitations.

site/ is the Docusaurus navigation shell over the canonical chapters.

book/build/ and PDF exports are generated artifacts and must stay out of Git.

The book's later capstone is a Book Intelligence Assistant. It will index this repository's chapters, research notes, experiments, and benchmarks; retrieve evidence with source paths; propose and critique improvements; and stop for human approval before any write. It does not modify the project autonomously.

What is implemented today

Chapters 1–3 have executable tensor, autograd, and PyTorch/MPS lessons with a recorded Day 1 benchmark. Chapters 4–5 now have a deterministic synthetic vision fixture plus matched PyTorch and TensorFlow/Keras CNN implementations. Chapters 6–7 add controlled transformer and MLX-LM observations. Chapters 8–9 now have a provenance-preserving local retrieval baseline and learned embedding path. Chapters 10–14 implement structured planning, persisted no-write approval, workflow comparison, and a deterministic reliability suite. All chapters are active manuscript drafts, not final editorial or print artifacts. The current project status and next task are maintained in docs/journals/from-tensors-to-agents-beta.md.

A setup contract before you install anything

The commands in this chapter follow a simple contract. The base development group is required for the shared tests. TensorFlow, Transformers, MLX, embeddings, and agent orchestration are optional learning milestones, each in a separate uv group. An optional group may download wheels, native libraries, or model weights into caches outside Git; it therefore has a different storage and network footprint from the source checkout.

Before any optional install, inspect the actual available filesystem with df -h .. There is no universal "enough disk" number: the right headroom depends on the package resolver, model cache, operating-system updates, and other work on the Mac. If the reported free space safely exceeds the expected download plus a comfortable working margin, install the group and record the result. If not, stop before the download, record the observed constraint, and continue with the deterministic or CPU-only portions of the book. Do not claim that an optional experiment ran merely because its code exists.

When a command fails, diagnose from the lowest-cost boundary outward. First check the command and working directory; then the Python version

and uv lock; then optional-group installation; then disk/cache state; then device/model availability; and only then the lesson's algorithm or framework. Preserve the first clear error and the command that produced it. Reinstalling every package or switching frameworks before identifying the boundary makes a local setup less reproducible, not more.

Prerequisites

Use a Mac with Apple Silicon when you want to reproduce MPS observations, but the early Python tests and examples are designed to fall back to CPU. Install:

Python 3.11; uv for Python environments and lockfile-based dependency installs; Node.js 20 and npm for the Docusaurus course; Pandoc for DOCX generation; and Microsoft Word on macOS for the **release** PDF export. LibreOffice is only a non-release fallback.

Before installing a large optional framework or model package, check available space with:

```
df -h .
```

Install it when there is sufficient headroom. If there is not, record the space constraint and pause that optional milestone rather than pretending it was completed.

Base installation

Clone the repository, enter it, and create the locked development environment:

```
git clone https://github.com/hassanvfx/mac-ai.git
cd mac-ai
uv sync --group dev
```

The exact public repository keeps every book command aligned with the course, the QR lab pages, and the live main branch. Clone it once, then run each chapter's commands from the mac-ai directory.

Validate the base environment:

```
uv run ruff check .
uv run pytest
uv run python experiments/01-tensors/broadcasting.py
uv run python experiments/02-gradients/autograd.py
uv run python experiments/03-pytorch/train_tiny_network.py
```

The last command chooses MPS when the installed PyTorch build reports it as available and otherwise uses CPU. An explicit device can be selected in the benchmark command:

```
uv run python benchmarks/01-day1/run.py --device auto --epochs 250 --runs 5 --seed 7
```

Recorded results are machine-specific evidence, not promises of performance on another Mac. Read benchmarks/01-day1/README.md before drawing conclusions.

First-session walkthrough

On a fresh machine, make the first session intentionally small. Run the base install, lint, tests, and the three Day 1 programs before adding a heavyweight framework. The expected outcome is not a particular MPS speed: it is a clean environment, numerical checks that pass, visible tensor/gradient output, and a tiny training run that selects MPS when it is available or clearly reports CPU fallback. This establishes that the repository, interpreter, and core package set agree before optional caches complicate diagnosis.

Then choose one branch of the learning path. Chapters 4–5 need TensorFlow only when you are ready to compare matched fixtures; Chapter 6 needs Transformers for tokenizer and classifier inspection; Chapter 7 needs MLX on Apple Silicon; Chapters 8–9 can use deterministic retrieval before learned weights; Chapters 10–14 can run their no-network fixture workflows without selecting an API provider. Every branch has an evidence record and a fallback. Completing an earlier branch is more useful than installing every optional group in advance.

At the end of a session, leave three artifacts: the command sequence that ran, the relevant benchmark or test result, and a concise journal note explaining what changed or what prevented progress. Generated caches, indexes, SQLite checkpoints, DOCX files, and PDFs remain local/ignored; canonical prose, source diagrams, experiment code, research notes, and benchmark descriptions belong in Git. This division is the foundation for both reproducible lessons and a clean future print release.

Optional TensorFlow/Keras installation

TensorFlow is isolated in an optional uv group so the foundation remains lightweight. Install and run the framework-comparison exercise only when you reach Chapters 4–5:

```
df -h .
uv sync --group tensorflow
uv run --group tensorflow python experiments/04-tensorflow/train_keras_cnn.py --epochs 30
```

Run the matching PyTorch program with the same declared fixture and training budget:

```
uv run python experiments/05-vision/train_pytorch_cnn.py --device auto --epochs 30
```

Compare correctness using the held-out accuracy, confusion matrix, and mistake list in benchmarks/02-vision/README.md. Do not compare the two elapsed values as framework speed: their current device and timing conditions are not a fair benchmark.

Optional Transformer installation

Chapter 6 uses Hugging Face Transformers with the existing PyTorch install. The library is an opt-in group; the first run also downloads a compact pretrained classifier into the local Hugging Face cache, outside this repository. Check space first and then install it:

```
df -h .
uv sync --group transformers
uv run --group transformers python experiments/06-transformers/inspect_sentiment.py --device auto
```

The experiment selects MPS when available and falls back to CPU. It prints the tokens, IDs, attention mask, manual logits-to-probabilities result, and the equivalent pipeline result. The current model cache observed on the project Mac is about 256 MiB, but model sizes vary widely; read benchmarks/03-transformers/README.md before treating it as a storage estimate.

Optional MLX and MLX-LM installation

Chapter 7 adds a local 4-bit instruction model through MLX and MLX-LM. These packages are Apple Silicon-specific and are kept out of the base environment:

```
df -h .
uv sync --group mlx
uv run --group mlx python experiments/07-mlx/run_local_model.py
```

The initial model download is approximately 276 MiB in the local Hugging Face cache. The script prints the exact model, declared 4-bit quantization, prompt, token counts, a warm-up, timing boundary, generation rate, and partial memory observations. It does not compare runtimes or rate answer quality; see benchmarks/04-mlx/README.md for the interpretation rules.

Optional learned-embedding installation

Chapters 8–9 add semantic search and evidence-only RAG over this repository. The small deterministic baseline remains available for tests and offline inspection; the learned path uses a compact Sentence Transformers encoder. Its first use downloads public model weights into the local Hugging Face cache, not the repository:

```
df -h .
uv sync --group embeddings
uv run --group embeddings python experiments/08-embeddings/book_search.py \
  --query 'Where do we record benchmark timing limitations?'
uv run --group embeddings python experiments/09-rag/grounded_answer.py \
  --query 'What should an experiment record?'
uv run --group embeddings python evals/run_book_intelligence.py
```

Use --deterministic with either program to exercise the lightweight baseline without loading the optional model. The RAG demonstration returns retrieved excerpts, source paths, and citation keys where available. It refuses a query when retrieval is empty or below its conservative threshold; it does not claim to produce a complete natural-language answer. Read benchmarks/05-book-intelligence/README.md for the recorded environment and limitations.

Optional structured-system installation

Chapter 10 compares the direct OpenAI-compatible SDK path with LangChain over the same retrieved evidence. The installed package group does not call an API or require a key; its default example is a no-network fixture comparison:

```
df -h .
uv sync --group agents
uv run --group agents python experiments/10-systems/compare_structured_planning.py
uv run --group agents python experiments/11-langgraph/approval_workflow.py
```

An intentionally separate --api mode requires all three process environment variables: BOOK_INTELLIGENCE_API_KEY, BOOK_INTELLIGENCE_API_BASE, and BOOK_INTELLIGENCE_MODEL. Never place their values in project files. A missing value fails before a request is made. See benchmarks/06-structured-systems/README.md for the narrowly recorded adapter comparison and its limits.

Site and manuscript builds

Build the course site with its own Node lockfile:

```
cd site
npm ci
npm run build
cd ..
```

Build the draft manuscript DOCX from the committed Lulu 6×9 Word template:

```
./scripts/build-book.sh
```

The course site can render the editable SVG diagrams directly. Pandoc needs an SVG rasterizer such as rsvg-convert available on PATH to embed those same figures in DOCX; otherwise it warns and omits the image. Install that print dependency before treating a DOCX proof as visually complete, then rebuild and inspect the generated pages. Keep the SVG source under book/assets/; any PNG fallback should be generated from that source at print resolution, not edited independently.

On macOS with Homebrew, install and verify that dependency with:

```
brew install librsvg
rsvg-convert --version
```

This writes book/build/from-tensors-to-agents.docx. It is a review artifact, not an upload-ready PDF. At print-production time, export it through Word on macOS, run the preflight script, and inspect every rendered page before calling the result release-ready.

Optional ClineFlow journaling

Read The Author's Toolkit for the optional ClineFlow workflow, its portable OKF knowledge-bundle rationale, and a small journal exercise. This companion repository neither installs nor depends on ClineFlow at runtime.X

How to learn with the repository

For every implemented chapter:

Read the intuition and predict the result.

Run the listed program unchanged.

Change one declared condition—shape, seed, device, or data difficulty.

Keep the original observation; record the new condition and result separately.

Read the associated benchmark or research note before making a general claim.

The course site and print book share this Markdown source. Complete code and measured observations stay in the companion repository so the prose can remain readable without hiding the evidence.

Start the course on your Mac

QR code for the online course start page

Start here: https://hassanvfx.github.io/mac-ai/

Clone the repository, follow the ten-day roadmap, and use the chapter QR codes to move from each lesson to its runnable lab. Scan with your phone and use Safari Share → AirDrop to send the course link to your Mac.

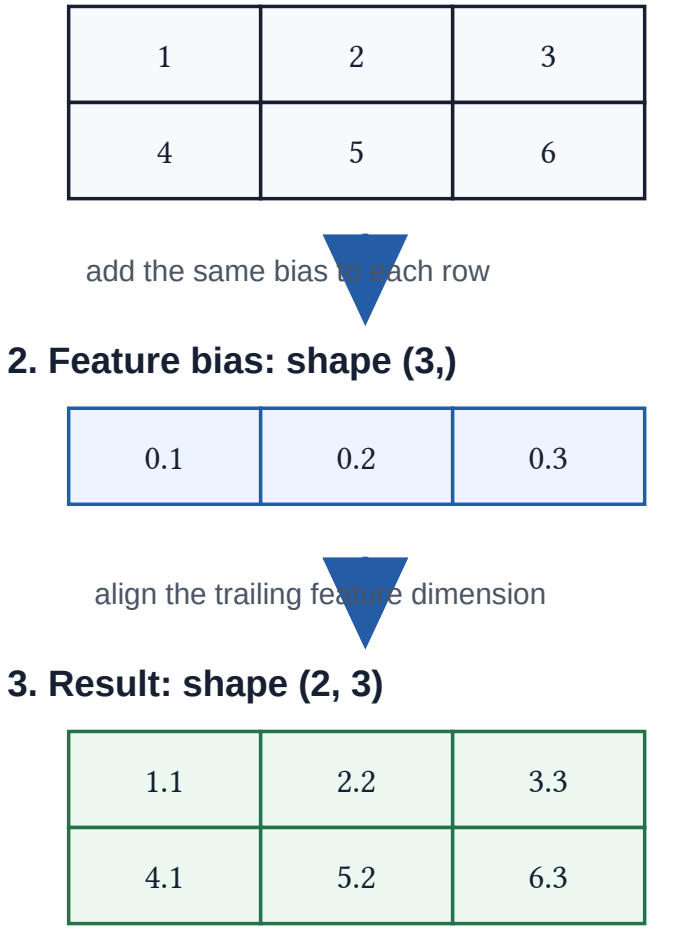
Everything Is a Tensor

Intuition

A tensor is a collection of numbers with a shape. The numbers carry values; the shape says how to interpret them. A scalar has shape (), a list of three measurements has shape (3,), a two-row table has shape (2, 3), and a batch of color images commonly has four axes: (batch, height, width, channels) or (batch, channels, height, width). The central habit of machine learning is to ask what every axis means before asking which model to use.

This is more than vocabulary. Linear algebra lets one operation describe many examples at once, and tensor libraries make that operation executable on CPUs and accelerators. The representation is therefore the bridge between a mathematical expression and a training program (Goodfellow et al. 2016).

The word *tensor* is not an instruction to force every object into a rectangle. It is a contract for the numerical part of a computation. Text, images, and documents begin in forms that humans can inspect; preprocessing selects properties and assigns them a numerical layout. That layout tells later code what operations are meaningful. Six values can form either two examples with three features or three examples with two features. The values are identical, but the program's question has changed.



A feature bias broadcasts across every example in a batch.

Problem

Suppose two examples each have three features and we want to add a different offset to each feature. We want one vector of offsets to apply to

every row, not a manual loop that can quietly confuse examples with features. The desired calculation is:

```
(2, 3) batch + (3,) bias ⟶ (2, 3) result
```

The second operand is *broadcast* across the leading batch axis. Broadcasting is concise, but it is also a common source of silent mistakes: (3,) means something very different from (3, 1).

PyTorch aligns dimensions from the right. They must match, one must be size one, or an absent leading dimension is treated as size one. That is why (2, 3) + (3,) works: features match and the bias expands across examples. (2, 3) + (2,) fails because the rightmost dimensions are three and two. This rule is useful, but it cannot know what an axis means. A valid broadcast over the wrong axis is often more dangerous than an exception because it leaves plausible numbers behind.

Keep batching and broadcasting separate. Batching gives one operation many independent examples. Broadcasting reuses a value along an axis. A learned bias is reused for every example; labels usually are not. This distinction reappears in losses, attention masks, image normalization, and embedding batches.

Minimal implementation

Run the complete example:

```
uv run python experiments/01-tensors/broadcasting.py
```

The program adds [0.1, 0.2, 0.3] to each row of a (2, 3) tensor. Predict the result before running it. The expected first row is [1.1, 2.2, 3.3]; if you instead expected each number in the first column to change by 0.1, you have correctly identified the feature axis.

The point of the exercise is not the arithmetic. It is the prediction. Before you run a tensor expression, name each value and its role: examples has shape (2, 3) and means two observations with three features each; bias has shape (3,) and means one offset per feature; shifted returns to shape (2, 3) and means the same two observations after the offset. This compact definition list prints more robustly than a table while preserving the shape contract.

This habit scales. For an image classifier, (32, 3, 224, 224) is not just a four-dimensional box: it means 32 images, RGB channels, and two spatial axes. For a language model, (batch, tokens, hidden) separates independent sequences from positions within a sequence and from the values used to represent each position. A model may accept an array with the right number of values and still produce nonsense if those axes have been exchanged.

Two operations that look similar have different meanings. reshape changes how a contiguous sequence of values is grouped; transpose or permute changes the order in which axes are interpreted. Neither operation discovers the intended meaning for us. Use a reshape when the grouping is known to be safe, such as turning a (batch, channels, height, width) image batch into a (batch, features) table immediately before a fully connected layer. Use a permutation when an API expects the same axes in another order. In both cases, write the before-and-after shapes and inspect a deliberately tiny example.

A worked axis audit

Consider two image batches that contain the same number of values. One has shape (2, 3, 4, 5) and is intended for a PyTorch convolution: two images, three channels, four rows, and five columns. The other has shape (2, 4, 5, 3) and is intended for a channels-last API: two images, four rows, five columns, and three channels. A permute(0, 2, 3, 1) converts the first meaning into the second. A reshape(2, 4, 5, 3) merely regroups adjacent values. It may produce the desired shape while scrambling which values belong to which channel and position.

This difference is worth testing with a recognizable tiny value. Make one image whose red, green, and blue channels contain different constants—say 10, 20, and 30. After a permutation, every pixel still has the same three channel values in a new axis order. After an unjustified reshape, those constants can appear interleaved across positions. Shape checks pass in both cases; checking one known location catches the semantic error. A good tensor test therefore asserts both shape and a few values with known meaning.

The same audit applies to language-model inputs. A token-ID tensor normally has shape (batch, sequence), while an embedding lookup returns (batch, sequence, hidden). Adding a positional embedding of shape (sequence, hidden) is usually intentional: it broadcasts across the batch. Adding a tensor of shape (batch, hidden) is not interchangeable. It either fails

or applies a per-example quantity across every token, which may be a different model. Write the intended expansion in words before relying on PyTorch's compact syntax.

Shapes, dtypes, and devices form one contract

Shape is only one part of a tensor boundary. A floating-point feature matrix, an integer class-label vector, a boolean attention mask, and a model parameter can have compatible dimensions while requiring different operations. For a classifier, logits often have shape (batch, classes) and floating-point dtype; targets have shape (batch,) and an integer class-index dtype. Cross-entropy uses those two meanings together. Converting targets to floating point merely to make an error disappear changes the loss contract rather than fixing it.

Device placement is part of the same sentence. inputs, model parameters, and any auxiliary tensors used in a computation need compatible devices. A CPU tensor and an MPS tensor can display the same values and shapes, but they cannot participate in one ordinary PyTorch operation. At a data boundary, record a small contract: name, shape, dtype, device, and semantic axes. This takes a few seconds and eliminates much of the guesswork behind common tensor errors.

A more realistic implementation

Neural-network batches use the same idea at larger scale. A linear layer maps an input batch X with shape (batch, input_features) to an output with shape (batch, output_features):

$$X @ W^T + b$$

Here b has shape (output_features,) and broadcasts once per example. The operation preserves the batch axis, replaces the feature axis, and gives one output vector per input. Writing those shapes beside the expression prevents many errors before Python runs.

There are three contracts to check at a layer boundary:

The final input axis must equal the layer's input_features.

Every leading axis represents independent items that should survive the operation unchanged.

The bias and any normalization parameters must align with the output feature axis, not accidentally with a batch or spatial axis.

For example, an input X of shape (4, 2, 3) can represent four documents with two tokens and three features per token. A linear layer with three input features can operate on it directly and returns (4, 2, output_features). It does not need a Python loop: PyTorch treats the leading axes as a collection of positions. That convenience is powerful, but only if the programmer has already decided that the two-token axis should be preserved.

Matrix multiplication supplies another useful boundary check. If X has shape (batch, features) and W has shape (outputs, features), then X @ W.T has shape (batch, outputs). The inner feature dimensions must agree; the batch dimension is not multiplied away. For a sequence tensor (batch, tokens, features), the same multiplication acts at every token and returns (batch, tokens, outputs). This is convenient when it is intended and dangerous when a reader thought tokens had been pooled already. Annotate the reduction or pooling operation that removes an axis; never assume a later linear layer did it for you.

Broadcasting can also create accidental memory pressure when code materializes a repeated tensor instead of using a view-like expansion. A bias with shape (features,) carries one value per feature. Repeating it into (batch, features) duplicates values and can be expensive for large batches, yet it usually changes no mathematical result. Prefer the smallest tensor that states the intended sharing rule. Profile only after the axis meanings are correct; optimizing a mistaken representation makes a wrong program harder to notice.

The companion experiment is intentionally small enough to modify and rerun in seconds: broadcasting.py. Later examples use the same contracts for a batch of training examples, per-example losses, and model parameters. If a later chapter seems mysterious, return to the layer boundary and annotate its axes first.

Experiment

Change the bias in experiments/01-tensors/broadcasting.py from (3,) to (2, 1). First predict the new result. Then deliberately try an incompatible shape such as (2,) and read PyTorch's error message. Record which axes you thought were aligned and which axes PyTorch attempted to align.

Extend the exercise with a second batch containing one example. Concatenate it with the original batch along axis zero, then verify that the bias still has shape (3,). This separates two ideas that beginners often conflate: changing the number of examples changes the batch axis, whereas changing the number of measurements per example changes the feature axis. A model that was trained for three features cannot silently accept four just because its batch size is allowed to vary.

For an observation worth keeping, record the command, PyTorch version, device, input shapes, expected output, and actual output. This first record establishes the format used by the repository's later benchmarks: an observation is more useful when another reader can reproduce the exact question it answers.

Add two negative tests to this exercise. First, assert that (2, 3) + (2,) raises a shape error; the failure confirms that PyTorch aligned dimensions from the right rather than guessing an axis. Second, construct a valid but wrong example: add a (2, 1) tensor to a (2, 3) batch and verify that it applies a different offset to each *example*, not each feature. The expected result is useful because it teaches the more dangerous case: successful execution is not evidence that the chosen axes match the intended mathematics.

Worked case: two valid broadcasts, one intended meaning

Start with the companion batch:

```
batch = [[1, 2, 3],
         [4, 5, 6]]          shape (2, 3)
```

The experiment's feature bias [0.1, 0.2, 0.3] has shape (3,). PyTorch aligns it with the rightmost axis and returns [[1.1, 2.2, 3.3], [4.1, 5.2, 6.3]]. That matches the declared story: feature zero receives 0.1 in every example, feature one receives 0.2, and feature two receives 0.3. The batch axis survives unchanged because the bias has no separate value for an example.

Now use a column-shaped tensor [[10], [20]] with shape (2, 1). It also broadcasts successfully, returning [[11, 12, 13], [24, 25, 26]]. The rightmost size-one axis expands across the three features; the leading size-two axis aligns with the two examples. This is correct arithmetic for a *per-example offset*, but wrong if the tensor was meant to be a per-feature bias. No error can reveal that mismatch because the operation is mathematically well defined.

Finally try [10, 20] with shape (2,). Right alignment compares its size two with the batch's trailing feature size three, so PyTorch raises a mismatch. The exception is useful documentation: the library has no basis to infer whether the two values describe examples, features, or something else. To make a per-example quantity explicit, write it as (2, 1); to make a per-feature quantity explicit, write (3,) or (1, 3). The extra axis is not decoration; it is the sharing rule made visible in the source.

This tiny three-way comparison is a debugging template. When a broadcast surprises you, create one example per leading axis and distinct constants per candidate meaning. Check the resulting values, not only the result shape. A shape assertion will accept both intended and unintended broadcasts; a recognizable-value assertion tells you which one actually happened.

What broke

The most useful early failure is a shape mismatch. Do not immediately reshape until the error disappears. State the intended meaning of each axis and verify that the reshape preserves it. A technically valid reshape can still be a conceptual bug—for example, mixing examples across a batch or treating image width as channels.

Other common failures are subtler. Calling squeeze() without an axis can remove the batch dimension when a final batch contains a single example. Using integer tensors where a differentiable floating-point calculation is expected can prevent a gradient from being created. Moving model weights to an accelerator while leaving inputs on the CPU produces a device mismatch. The remedy is not a list of magic reshapes; it is an invariant: at each important boundary, assert the dtype, device, rank, and named axis sizes you expect.

When debugging, shrink the problem before adding print statements everywhere. Use two examples and two or three features, print shapes rather than full large tensors, and compare one result with a hand calculation. Once the small case is understood, restore the real data. This discipline also makes tests clearer: a numerical test can show the expected values while a shape test protects the meaning of the operation.

When a library reports an error such as "size of tensor a must match tensor b," turn the message into an axis comparison. Write both shapes below one another, align them from the right, and label each axis. Ask

whether a size-one axis is an intentional reusable quantity or an accidental singleton created by slicing or unsqueeze. This method is slower than trying transformations at random for one minute, but it produces a reusable explanation instead of a fragile patch.

Alternatives and when to use them

Python lists work for tiny demonstrations, and NumPy uses the same broad array-and-broadcasting model. Use PyTorch tensors when the next step needs automatic differentiation, neural-network modules, or accelerator placement. Avoid a tensor library abstraction only when the data is truly irregular and a named record or graph structure communicates the domain better.

Named tensor tools, type annotations, and runtime shape-checking libraries can make the contracts more explicit. They are helpful in a large codebase, but they do not replace the underlying reasoning and can make a first experiment harder to read. Plain PyTorch shape comments are a good default for this book: they are visible beside the operation, work on every supported machine, and make the transition to another framework straightforward. TensorFlow, JAX, and NumPy all use closely related array semantics even though their module APIs differ.

Do not use broadcasting merely to avoid a line of code. An explicit reshape, such as bias[None, :], can be preferable when it documents the axis that is being expanded. Conversely, avoid repeating a bias to the full batch shape only to make the addition look explicit: it consumes memory and obscures the fact that the values are shared. Prefer the representation that makes the mathematical intention easiest for a future reader to verify.

Evidence trail

Read the working source note at research/01-tensors/notes.md, run experiments/01-tensors/broadcasting.py, and use tests/test_day1.py for the small numerical expectations that protect this lesson.

Takeaway

Tensors are not merely containers. Their shapes are part of the program's meaning. Before changing an operation, name every axis, write the input and output shapes, and decide exactly which axes may broadcast.

Run the lab on your Mac

QR code for Chapter 01 lab

Online lab: https://hassanvfx.github.io/mac-ai/labs/01

Repository: experiments/01-tensors/broadcasting.py

```
uv run python experiments/01-tensors/broadcasting.py
```

Expected: Tensor shapes and broadcasting output.

Evidence: benchmarks/01-day1/README.md

Scan the code to open the live lab, then use Safari Share → AirDrop to send it to your Mac. The lab follows main; clone the repository once and run the command from its root directory.

Learning Is Error Correction

Intuition

Learning begins with a disagreement: a model makes a prediction, a target says what should have happened, and a loss function turns their difference into one number. A gradient then answers a local question: if a parameter moves a tiny amount, how does that loss change? It is not a learning rule by itself. It is the measurement used by an optimizer to choose a corrective step (Goodfellow et al. 2016).

The useful mental model is a repeated correction cycle, not a machine that "understands" after one update. A model has adjustable parameters, a dataset supplies examples and targets, and a loss compresses the model's disagreement with those targets into a number the training process can minimize. Each step uses only local information: the gradient tells us which nearby changes would raise or lower the current loss. It does not prove that the next step improves unseen data, finds the global best solution, or produces a useful model. Those questions require a held-out evaluation and a careful experiment.

This distinction matters because loss can be small for the wrong reason. A model might memorize a tiny training set, exploit a data leak, or optimize an objective that fails to represent the real task. Gradient descent is extremely effective at following a stated objective; it cannot decide whether that objective was the right one. Treat a training loss as evidence about the training calculation, not as a general claim about quality.

Problem

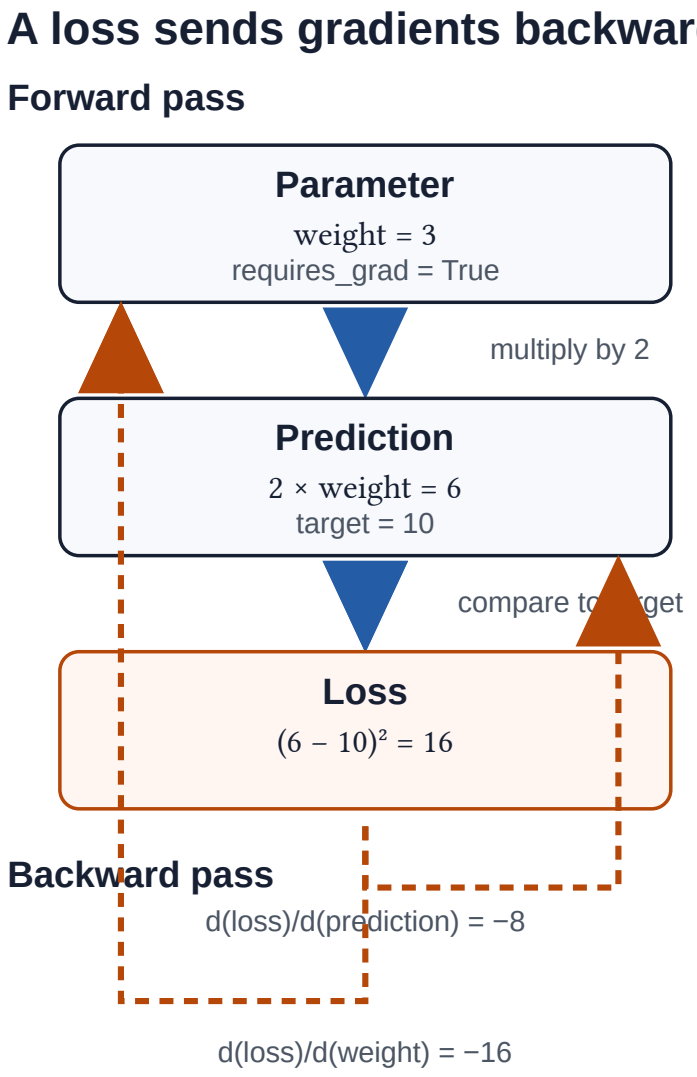
Take the one-parameter model prediction = 2 × weight, target 10, and squared error loss (prediction - target)². With weight = 3, the prediction is 6, the loss is 16, and the analytic derivative is:

$$d/d(\text{weight})\ (2 \times \text{weight} - 10)^2 = 4 \times (2 \times \text{weight} - 10) = -16$$

The negative value means that a small increase in this particular weight reduces the loss locally. The goal is to verify that PyTorch's autograd system constructs the same derivative from ordinary tensor operations.

The chain rule makes the calculation less mysterious. Let r = prediction - target, so the loss is r^2. A small change in weight changes prediction by two times that amount; changing r changes the squared loss by 2r. At weight = 3, residual r is −4, so the local rates multiply to $2 \times (-4) \times 2 = -16$. Backpropagation is this bookkeeping repeated through a longer computation graph. It is not a second model that discovers a learning rule; it evaluates derivatives of the forward function already defined.

For a vector of parameters, the gradient has one component per parameter. If $w = (w_1, w_2)$, then $\nabla L = (\partial L/\partial w_1, \partial L/\partial w_2)$ describes the local slope in each coordinate. Gradient descent subtracts that whole vector. A component near zero does not mean its feature is unimportant; it means this loss is locally insensitive to that parameter at this point.



The forward pass computes a loss; the backward pass propagates its gradient to the parameter.

Minimal implementation

```
uv run python experiments/02-gradients/autograd.py
```

The expected output is a prediction of 6.0, a loss of 16.0, and d(loss)/d(weight)=-16.0. The automated test independently verifies that analytic result in tests/test_day1.py.

A more realistic implementation

Real models contain many parameters and nested functions. Rather than writing one derivative per parameter, PyTorch records the operations used to produce a loss. Calling loss.backward() applies the chain rule through that recorded computation. An optimizer then reads each parameter's .grad value and updates it. The common training-loop order is:

```
zero old gradients → forward pass → loss → backward pass → optimizer step
```

The order matters. Gradients accumulate by default because accumulation is useful for some workloads. In ordinary minibatch training, failing to clear them combines the current gradient with stale work from previous steps.

For a scalar parameter w, a basic gradient-descent update is:

```
w ← w - learning_rate × d(loss)/d(w)
```

In the example, the derivative is −16. With a learning rate of 0.1, the new weight is 3 - 0.1 × (−16) = 4.6: the parameter moves upward because an upward movement reduces this loss nearby. The sign is the important part. The optimizer subtracts the gradient because the gradient points toward increasing loss. A learning rate decides how far to walk in the opposite direction.

Learning rate is a trade-off, not a universal constant. If it is too small, training makes little visible progress. If it is too large, the update can overshoot a useful region and make the loss oscillate or diverge. The right value depends on input scaling, architecture, loss, optimizer, batch size, and the data itself. Start with a documented baseline and examine the loss curve rather than interpreting a single final number.

The repository's tiny regressor makes this cycle concrete. It trains a deliberately small network using mean squared error and SGD; its recorded Day 1 observation is available in the companion repository's benchmarks/01-day1/README.md record. That record reports a particular workload and machine configuration. It is not a promise that every MPS or CPU workload will have the same timing or memory behavior.

A worked minibatch view

Most training code evaluates several examples at once. If a batch contains predictions $p_1, \ldots, p_n$ and targets $y_1, \ldots, y_n$, mean squared error is often written as:

$$\text{loss} = (1 / n) \times \Sigma_i (p_i - y_i)^2$$

The reduction to one scalar is important. Reverse-mode autograd efficiently computes derivatives of one scalar loss with respect to many parameters. The mean also makes the scale of the loss less sensitive to batch size than a raw sum, although changing batch size still changes the noise and frequency of updates. Classification chapters use cross-entropy instead of squared error, but the forward → loss → backward → update rhythm remains the same.

In PyTorch, tensors produced by operations remember enough of their history to differentiate when needed. Leaf tensors such as a learnable weight receive a .grad after backward(). Intermediate predictions usually do not need to store a user-visible gradient. This is why a training loop reads gradients from model parameters, not from every temporary result. It is also why retaining large computation graphs accidentally can consume significant memory.

Reduction defines the scale of a gradient

Suppose two identical examples appear in a batch. With a summed squared-error loss, duplicating the batch doubles both the loss and its gradient. With a mean loss, duplicating the batch leaves their expected scale approximately unchanged. Neither choice is universally correct, but it changes the effective learning rate a reader experiences. When a script switches from a mean to a sum, or changes batch size, record it before concluding an optimizer became unstable.

Autograd normally needs a scalar output for backward() because reverse mode answers “how does this one scalar objective change with every parameter?” A vector-valued result can still be differentiated if code supplies an upstream vector, but that is a different contract. In ordinary supervised learning, reducing per-example errors to one declared loss makes the target and gradient scale visible.

Reading a gradient without overinterpreting it

A negative scalar gradient in the opening example tells us that increasing the weight slightly lowers the current loss. It does not tell us that increasing the weight forever is safe. At weight = 5, the prediction reaches the target and the gradient is zero. On the other side, the residual changes sign and the gradient becomes positive, so subtraction moves the weight back down. This is a local feedback signal in a simple quadratic bowl, not a general promise about every neural-network loss surface.

Experiment

Change the starting weight and target in the autograd script. Calculate the derivative on paper before execution, then compare it with weight.grad. Next, call backward() twice without clearing the gradient and observe the accumulation. Finally, reset the gradient and explain why the normal training loop performs that reset before every forward pass.

Then turn the single calculation into five manual updates. After each update, write down weight, prediction, loss, and gradient. You should see the loss fall for a sensible learning rate, while the size of the gradient typically shrinks near a local minimum. Repeat with a much larger learning rate. If the loss rises or jumps between values, do not call the model broken: you have observed an optimizer configuration that takes steps too large for this local surface.

Finally, compare autograd with a finite-difference estimate. For a small positive ε, approximate the derivative by:

```
(loss(weight + ε) - loss(weight - ε)) / (2 × ε)
```

It should be close to weight.grad for a carefully chosen epsilon. It will not be exactly equal because floating-point arithmetic and the approximation both introduce error. This comparison is valuable when implementing a custom operation: finite differences are slow, but they can reveal whether a gradient is wildly wrong before a large training run hides the source of the problem.

Choose epsilon with care. If it is too large, the finite difference measures a wide nonlinear interval instead of the local derivative. If it is too small, the two floating-point loss values can round together and make subtraction noisy. Try a small range of epsilons and look for agreement over a stable region. The scalar exercise is ideal because its analytic answer is known; for

a custom layer, compare only a few chosen parameters so the diagnostic stays inexpensive and understandable.

Add a reduction experiment as well. Compute squared errors for two examples, then compare .sum() and .mean() losses and their gradients. Duplicate the same examples and repeat. Write down which values double and which remain stable. This turns a framework default into a visible mathematical choice that matters later when batch sizes and metrics change.

Worked case: five corrections toward one target

Use the opening scalar model with target 10, initial weight 3, and learning rate 0.1. Its gradient is 4 × (2 × weight - 10), and the update subtracts 0.1 times that quantity. The first five pre-update states are:

step	weight	prediction	loss	gradient
0	3.00000	6.00000	16.000000	−16.00000
1	4.60000	9.20000	0.640000	−3.20000
2	4.92000	9.84000	0.025600	−0.64000
3	4.98400	9.96800	0.001024	−0.12800
4	4.99680	9.99360	0.000041	−0.02560

The numbers make three ideas visible. First, the negative gradient produces an upward weight update because gradient descent subtracts it. Second, the loss falls rapidly here because this is a simple smooth quadratic and the chosen learning rate is conservative. Third, the gradient magnitude shrinks as the prediction approaches the target; it is feedback about the current parameter, not a fixed instruction that the weight should always increase.

Now repeat the calculation with a learning rate of 1.0. The first update moves from 3 to 19, producing a prediction of 38 and a much larger loss. The next gradient points back in the other direction, so the weight overshoots again. This does not contradict autograd: the derivative is still correct at each point. The optimizer configuration uses a step too large for this curvature. Separating “the gradient is wrong” from “the update rule is unstable” is one of the most useful early debugging distinctions.

The test in tests/test_day1.py anchors the opening derivative at −16. A finite-difference check would approximate the same local slope, while the five-step table checks the update arithmetic. Neither establishes that a neural network will generalize; it establishes that a small declared objective, derivative, and update are internally consistent before more data and layers obscure the calculation.

What broke

Two beginner errors reveal important boundaries. A tensor without requires_grad=True does not request a derivative with respect to itself. And calling backward() after discarding or mutating a computation graph can produce an error because autograd needs the original operations. The repair is not to memorize error messages; it is to identify which values are parameters, which values are observations, and which loss produced the gradient.

Another failure is reading a gradient after the optimizer has already updated the parameter and assuming it describes the new model. A gradient belongs to a specific forward pass and parameter value. Record the loss and gradient before the update when debugging. Similarly, in-place mutation of a value autograd needs can invalidate its saved history. Prefer ordinary expressions first; introduce in-place operations only when profiling shows a real need and tests protect the computation.

Numerical scale can also fail quietly. Extremely large activations or a poorly chosen exponential can create inf or nan, which then flows through the loss and gradients. Add checks for finite loss values in experiments, inspect input ranges, and reduce the problem to the first batch that fails. Gradient clipping, normalization, and more stable loss implementations are tools for particular problems, not substitutes for identifying where the invalid value first appeared.

Detaching tensors is another quiet failure. Calling .item(), converting to NumPy, or using .detach() at the wrong point removes a value from the graph. That is correct for logging a finished loss, but wrong for a value that must still influence a later differentiable computation. Keep logging and model calculation separate: compute the loss, call backward(), then detach only the small values intended for display or storage.

Alternatives and when to use them

For a one-variable function, symbolic or hand differentiation is clearer and often safer. Finite differences are useful as a slow diagnostic check. Use reverse-mode automatic differentiation when one scalar loss depends on many parameters—the standard setting for neural-network training.

Forward-mode automatic differentiation can be attractive when there are few parameters and many outputs, which is the opposite shape of most

neural-network training. Optimization methods also differ: SGD is transparent and provides a useful baseline; momentum smooths updates; adaptive methods such as Adam adjust per-parameter step scales. No optimizer removes the need for a validation metric and an explicit stopping rule. Use the simplest optimizer that makes the experiment readable, then compare alternatives under the same data split, seed, and measurement method.

Evidence trail

The gradient source note is research/01-tensors/notes.md; the runnable scalar example is experiments/02-gradients/autograd.py, with deterministic checks in tests/test_day1.py and the Day 1 record in benchmarks/01-day1/README.md.

Takeaway

Loss measures error, gradients measure local sensitivity, and an optimizer chooses the update. Keeping those roles separate makes backpropagation understandable rather than magical.

Run the lab on your Mac

QR code for Chapter 02 lab

Online lab: https://hassanvfx.github.io/mac-ai/labs/02

Repository: experiments/02-gradients/autograd.py

```
uv run python experiments/02-gradients/autograd.py
```

Expected: Analytical and autograd gradients agree.

Evidence: benchmarks/01-day1/README.md

Scan the code to open the live lab, then use Safari Share → AirDrop to send it to your Mac. The lab follows main; clone the repository once and run the command from its root directory.

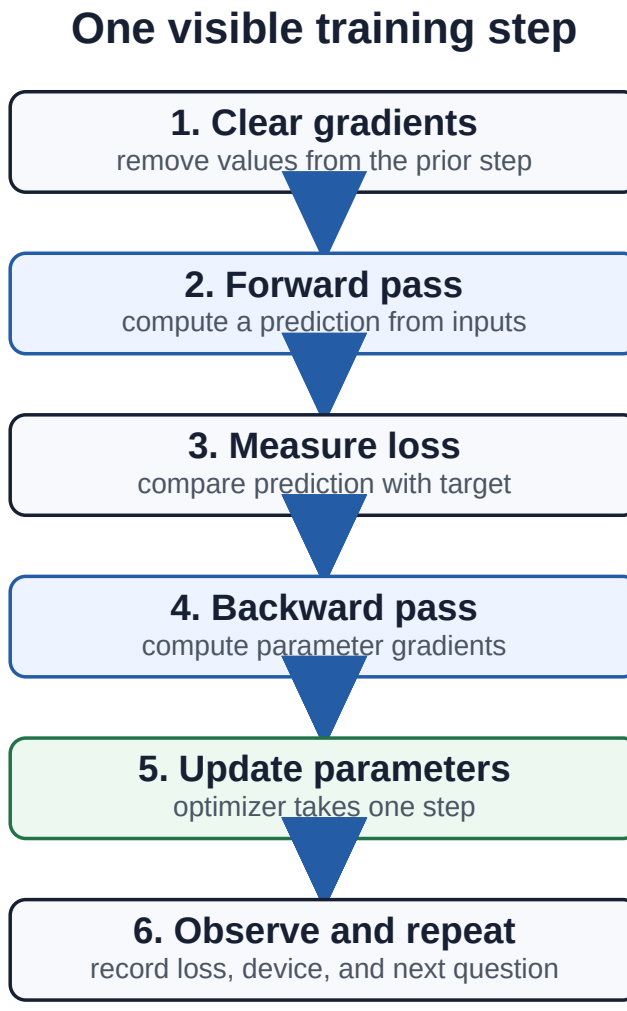
Building a Neural Network with PyTorch

Intuition

A neural network becomes useful only when its parameters participate in a repeatable cycle: make a prediction, measure error, compute gradients, and update the parameters. PyTorch exposes that cycle directly. That visibility is valuable at the start because it lets us connect the tensor shapes from Chapter 1 and gradients from Chapter 2 to an actual learning system.

There are two complementary ways to read a PyTorch program. At the mathematical level, a network is a function with parameters: inputs go in, predictions come out, and training adjusts the parameters to reduce a loss. At the program level, nn.Module groups those parameters and its forward method describes the calculation. The optimizer does not know the meaning of a layer; it simply receives the module's registered parameters and updates their gradients after the backward pass. Keeping those roles separate makes it easier to replace one part without losing the training contract.

The goal of this chapter is not to claim that a tiny network resembles a production model. It is to make every moving part visible under conditions that can be repeated. A small synthetic task lets us inspect shapes, loss values, device selection, and seed behavior without a download, a lengthy run, or a dataset whose quirks dominate the lesson.



The explicit six-step PyTorch training loop.

Problem

Build the smallest network that can learn a known relationship without hiding the training loop behind a high-level trainer. The fixed task is 64 synthetic pairs generated from target = 2 × input + 0.5. The model is:

```
Linear(1, 8) ⟶ Tanh ⟶ Linear(8, 1)
```

This network has more capacity than the linear target requires. That is intentional: it demonstrates the standard PyTorch components while retaining a small, deterministic workload whose behavior is easy to inspect.

The input tensor has shape (64, 1): 64 independent examples and one scalar feature per example. Targets have the same shape. The first linear layer turns each one-feature example into eight hidden values, Tanh applies the same nonlinearity to each hidden value, and the final layer maps eight hidden values back to one prediction. The batch axis is preserved at every stage:

```
(64, 1) ⟶ Linear(1, 8) ⟶ (64, 8) ⟶ Tanh ⟶ (64, 8) ⟶ Linear(8, 1) ⟶ (64, 1)
```

This is deliberately more flexible than necessary. A single linear layer could represent the target relationship exactly. The hidden layer introduces the composition used by larger networks while the known target makes it possible to notice an implausible result quickly. If this model cannot reduce mean squared error, the first suspects are the data shape, seed, update order, learning rate, or device—not the difficulty of the task.

The task gives us a useful sanity check. Because the target is known, 2 × input + 0.5, a model prediction can be compared with both the training loss and the underlying relationship. The hidden layer is intentionally overcomplete for this job: it can represent the mapping, but also makes the parameter path less obvious than a single slope and intercept. That makes it a good bridge from Chapter 2's scalar gradient to a module with many weights.

Count the contracts before running code. The dataset fixes input/target shapes and deterministic construction. The module fixes input width, hidden width, activation, and output width. Mean squared error fixes the objective. SGD fixes how gradients become updates. Device choice fixes where compatible tensors execute. If loss fails to fall, change and record one contract at a time rather than changing model width, learning rate, seed, and device together.

Minimal implementation

Run the complete training example:

```
uv run python experiments/03-pytorch/train_tiny_network.py
```

The script selects Apple MPS when the installed PyTorch build reports it as available; otherwise it falls back safely to CPU. It fixes the random seed to 7, trains for 250 epochs with SGD and mean squared error, and prints the first and final losses.

Read the script beside its shared training function before running it. The entry point chooses a device, but the function responsible for training receives that device explicitly. It then fixes the PyTorch seed, moves both inputs and targets to the selected device, creates the model on that device, and repeats the training loop. Passing the device rather than relying on a global default keeps the CPU and MPS paths comparable: the calculation is identical apart from where compatible operations execute.

The loop contains a few details that are easy to overlook:

model.train() is implicit for this tiny model because it has no dropout or batch-normalization layers, but real training scripts should set their mode deliberately.

optimizer.zero_grad() happens before backward() because PyTorch adds new gradients to existing .grad tensors by default.

loss.detach().cpu().item() records a plain Python number. Detaching avoids retaining every epoch's computation graph; moving to CPU allows a scalar to be read consistently when the model is on MPS.

The loop stores a loss for inspection, but it does not use that training loss to decide that the model is ready for an unseen dataset.

Those choices generalize. The later CNN and transformer examples add data loaders, validation metrics, and more complex modules, but the core feedback cycle remains explicit.

Parameters are data owned by the module

nn.Module is more than a convenient container. When a layer is assigned as an attribute, PyTorch registers its learnable tensors so model.parameters() can hand them to the optimizer. A plain tensor created inside forward is not a durable parameter merely because it participates in arithmetic. Create intended parameters in __init__, use them in forward, and verify their shapes before training.

The optimizer stores state about the parameters it received. In this SGD example, that state is minimal; an optimizer such as Adam also keeps moving averages. Replacing a model parameter after constructing an optimizer can leave the optimizer updating the old tensor. Keep model construction, optimizer construction, and the training loop in visible order.

Loss curves are debugging traces, not release criteria

The loss list is a trace of one optimization run. A falling curve tells us that this model is reducing this objective on these examples. It does not certify generalization, calibration, or stability outside the observed interval. A flat initial curve can suggest an update-order mistake, zero/invalid gradients, unsuitable learning rate, or a device/data mismatch. A falling curve followed by a rise can suggest overshooting. Print a few checkpoints before relying only on the last number.

Real implementation: make the observation reproducible

The benchmark runner makes device selection, warmup, synchronization, number of timed runs, and seed explicit:

```
uv run python benchmarks/01-day1/run.py --device auto --epochs 250 --runs 5 --seed 7
```

On the recorded M4 Pro environment (macOS 26.1, Python 3.11.9, PyTorch 2.13.0), MPS was available. Five timed runs reached the same final loss, 0.000994, from an initial loss of 1.892602; their median wall time was 120.537 ms. The complete workload, individual values, and limitations are recorded at benchmarks/01-day1/README.md in the companion repository.

This is evidence about one tiny synthetic workload on one machine. It is not a claim that MPS is faster than CPU, nor a prediction about larger networks.

Reproducibility has layers. Fixing a seed makes this compact workload repeat on the recorded environment, but it does not make every accelerator operation, library version, or data-loader configuration deterministic. Record the environment instead of promising absolute repeatability: operating system, Python and PyTorch versions, selected device, model, dataset construction, optimizer, epoch count, warmup procedure, and timing method

all affect what a benchmark means. A useful benchmark answers one narrow question well.

Device selection deserves the same discipline. MPS is Apple's Metal backend for PyTorch-compatible operations, but availability is not enough to prove that every operation in a future model is supported or optimal there. The helper returns CPU when MPS is not available, so the educational program remains runnable on other machines. An explicit MPS request, by contrast, should fail when unavailable; silently changing a requested benchmark device would make a comparison ambiguous.

Experiment

Run the benchmark once with --device cpu and, on an MPS-capable Mac, once with --device mps. Do not compare only the median number: first confirm the same seed, epochs, versions, warmup procedure, and final loss. Then increase the epoch count and observe whether setup time becomes a smaller fraction of the measurement. Record every changed condition rather than overwriting the existing observation.

Before comparing devices, use the same code path and verify the result rather than only the elapsed time. The final loss should be compatible with the fixed workload, and the selected device should appear in the printed output. Then separate setup from steady-state work: first model creation, allocator setup, and compilation-like initialization can dominate a microbenchmark. The runner uses one unmeasured warmup and measures multiple timed runs to make that choice visible. It still does not report a CPU comparison result until one is actually run and committed.

Try one controlled failure as well. Request --device mps on a CPU-only machine, or temporarily request an unavailable backend in a unit test. The right outcome is a clear error that tells the reader what was requested and how to use the safe fallback. A benchmark that quietly changes the requested condition produces a polished but misleading number.

The benchmark's five timed values deserve the same care as a loss trace. The median summarizes five repeats after one unmeasured warm-up; it does not reveal cold-start experience, distribution tails, energy use, or memory use. If a future question concerns first-use latency, measure model creation and the first training call separately. If it concerns steady state, retain the warm-up and show variation. Do not edit the existing record to make a later

measurement look comparable—create a new record with its own workload and date.

MPS synchronization is a measurement boundary rather than a performance trick. Device work can be queued; a host timer that stops before the queue completes reports Python dispatch time rather than complete model work. Synchronizing before and after the timed section answers a clearer wall-clock question for this runner. Other asynchronous pipelines need their own declared boundaries.

Worked case: one tiny run, three different questions

Run the teaching script with its fixed seed, 250 epochs, SGD learning rate 0.08, and automatic device selection. On the recorded M4 Pro observation, the first loss was 1.892602 and the last was 0.000994. That answers the first question: under this declared synthetic dataset and model, the explicit loop reduced its training mean-squared error. It does not answer whether the model would generalize, because the exercise deliberately has no held-out dataset.

The second question is reproducibility. The benchmark repeats the same workload five times after an unmeasured warmup. Each recorded MPS run reaches the same final loss, 0.000994. That provides evidence for deterministic behavior of this narrow seed/environment/workload combination. It does not mean every PyTorch program is deterministic on every accelerator, nor that an edited model with dropout, shuffled data, or different versions will reproduce the number automatically.

The third question is timing. The five recorded wall-clock values range from 119.637 to 122.571 ms, with a median of 120.537 ms after MPS synchronization. The timer includes the declared training workload and excludes the unmeasured warmup; it is not a CPU comparison, first-use latency result, memory measure, or proxy for a larger network. The three questions need three sentences in a benchmark record because “the model trained in 120 ms” would collapse convergence, repeatability, and timing into one misleading claim.

To debug a run that does not follow this pattern, inspect contracts in order. Confirm that inputs and targets retain shape (64, 1), that the model and both tensors share the selected device, that zero_grad precedes backward, and that the seed is set before model construction. Then lower the learning rate or shrink the program to one batch. Do not first add layers or

rerun until a favorable final loss appears; a reproducible failure is the evidence needed to locate the broken training boundary.

What broke

Accelerator availability is conditional. The example must run on CPU-only machines, and an explicit --device mps request fails clearly if MPS is unavailable. Also, accelerator work may be queued: timing without an MPS synchronization can stop the clock before all device work finishes. The benchmark runner synchronizes before and after each timed run for that reason.

The most common model failure is a device split: model parameters live on MPS while inputs or targets remain on CPU. The resulting error is useful because it identifies the boundary. Move the complete batch—inputs, targets, and any attention masks or auxiliary tensors—to the same selected device as the model. Do not solve it by scattering .to("mps") calls throughout a network; choose a device once, pass it inward, and make the transfer visible at the data boundary.

Another tempting mistake is to measure an improving training loss and call the network accurate. This synthetic exercise has no held-out split because its purpose is to test the feedback loop, not generalization. In a real task, keep validation and test examples separate from parameter updates. Later vision chapters will use that separation for metrics and error inspection.

Overfitting is deliberately invisible here because no unseen-distribution claim is being made. The vision experiment adds validation and test partitions because its question changes from “does the loop reduce a known loss?” to “does this classifier solve a held-out fixture?” Reusing training loss as a quality score across those questions would blur the distinction this regressor teaches.

Alternatives and when to use them

For a first prototype, an explicit loop is the best choice because every step is inspectable. Higher-level PyTorch training frameworks can reduce repetition after the fundamentals are understood. For a strictly linear relationship, a single Linear(1, 1) model is simpler; the small hidden layer here exists to teach composition, not because the data demands it.

NumPy plus hand-written gradients is a reasonable way to learn the mechanics, but it becomes fragile as a model gains branches and many parameters. TensorFlow and JAX provide other automatic-differentiation ecosystems; Chapter 4 compares an equivalent vision task rather than treating API syntax as a framework benchmark. Within PyTorch, a high-level trainer can manage checkpoints, mixed precision, and distributed execution after an explicit baseline has established the data, loss, metric, and failure behavior. Abstraction should remove repetition, not conceal the evidence needed to trust an experiment.

Mixed precision, gradient accumulation, schedulers, checkpoints, and data loaders are valuable later additions, but each adds a contract. Mixed precision changes numeric formats and loss scaling; accumulation changes the effective batch/update schedule; checkpoints add versioning and restoration questions. Introduce them only when the baseline's data, model, loss, metric, and device boundaries already have tests. A small explicit loop remains the reference implementation against which automated training can be checked.

Evidence trail

Read research/01-tensors/notes.md, run experiments/03-pytorch/train_tiny_network.py, and interpret timing only through benchmarks/01-day1/README.md and its declared workload.

Takeaway

PyTorch training is a controlled feedback loop. A useful result is not merely a lower loss: it is a lower loss tied to a known seed, dataset, model, device, measurement method, and record of what the result does *not* prove.

Run the lab on your Mac

QR code for Chapter 03 lab

Online lab: https://hassanvfx.github.io/mac-ai/labs/03

Repository: experiments/03-pytorch/train_tiny_network.py

```
uv run python experiments/03-pytorch/train_tiny_network.py
```

Expected: MPS is selected when available; CPU otherwise.

Evidence: benchmarks/01-day1/README.md

Scan the code to open the live lab, then use Safari Share → AirDrop to send it to your Mac. The lab follows main; clone the repository once and run the command from its root directory.

PyTorch vs TensorFlow: Same Neural Network, Two Philosophies

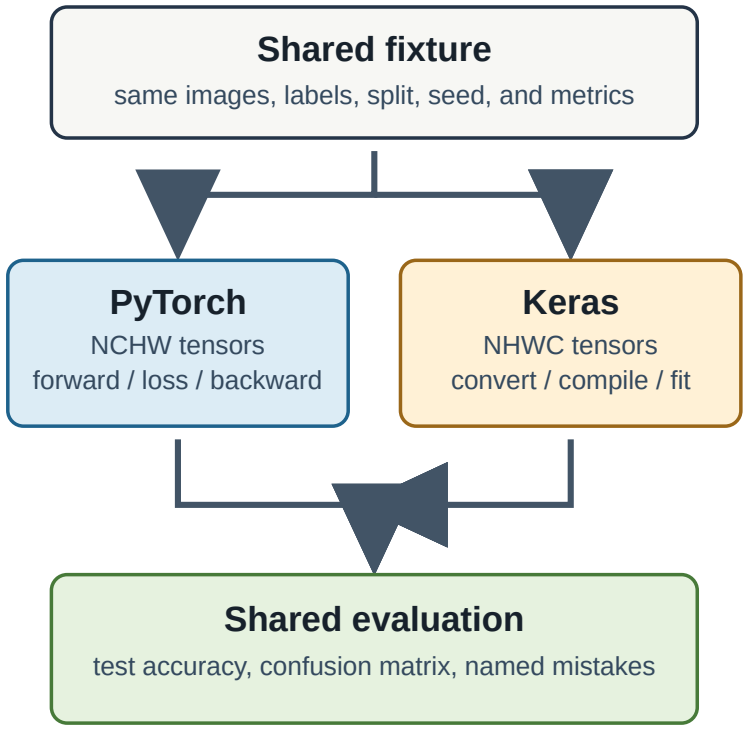
Intuition

The model is mathematical; the framework determines how directly we express, inspect, and deploy that mathematics. A convolution, a cross-entropy loss, and Adam do not become different ideas because their APIs differ. What does differ is where the framework places its boundaries: explicit tensors and loops in PyTorch, or composable layers and a training interface in Keras.

The useful comparison is therefore not a contest of short code samples. It is one learning question implemented twice with the data split, seed, topology, metric, and limits declared in advance.

That discipline protects against an appealing but weak comparison: placing two unrelated tutorials side by side and judging whichever one has fewer lines. A framework program includes data layout, initialization, batching, device selection, defaults, and measurement boundaries as well as model layers. If those differ, an observed difference may come from the experiment rather than from the framework. The comparison here is intentionally narrow enough to inspect, then clear about the questions it cannot answer.

Think of the comparison as a checklist of invariants. The class meaning, fixture generator, split seeds, labels, topology, optimizer family, learning rate, epoch budget, and evaluation outputs belong to the problem contract. The tensor layout, loop API, initializer implementation, batching mechanism, device visibility, and history object are framework or runtime choices. Separating these categories is useful whenever results differ: inspect a violated invariant before attributing the change to a framework philosophy.



Correctness comparison, not a speed leaderboard.

The shared task branches only where framework presentation requires it, then converges on one held-out evaluation contract.

The visual is a practical review tool. Start at the shared fixture and trace every arrow. A value may change layout when it crosses into Keras, but its class meaning, split membership, and evaluation label must not change. A training loop may become a fit call, but the declared optimizer, loss, and epoch budget must remain recognizable. The branches may differ in presentation; they must not silently define different learning questions.

Problem

Train the same small convolutional classifier in PyTorch and TensorFlow/Keras without changing the fixture, evaluation question, or model shape. The task is to classify 16×16 synthetic images containing vertical, horizontal, or diagonal strokes. It is intentionally a controlled teaching fixture, not an image benchmark (Goodfellow et al. 2016).

Minimal implementation

The two runnable implementations are:

```
uv run python experiments/05-vision/train_pytorch_cnn.py --device auto --epochs 30
uv sync --group tensorflow
uv run --group tensorflow python experiments/04-tensorflow/train_keras_cnn.py --epochs 30
```

Both use the deterministic fixture in src/from_tensors_to_agents/vision.py. They train the same sequence of operations: two 3×3 convolutional layers with 8 then 16 channels, ReLU activations, pooling, global average pooling, and a three-output classifier. Both use Adam with learning rate 0.01, training seed 17, and 30 epochs. PyTorch keeps the optimization loop visible; Keras encapsulates it in model.fit while keeping the model definition readable.

TensorFlow is an opt-in project group, not a requirement for the first three chapters. The official TensorFlow installation guide is the authority for supported platform details (TensorFlow 2026).

The two APIs make different trade-offs in visibility. In the PyTorch program, the reader can see the forward call, loss, backward(), and optimizer update in the source. This makes it easy to insert a diagnostic or change the training rule. In the Keras program, compile declares the optimizer and loss, and fit owns the standard loop. This removes routine code and collects a training history, but custom behavior moves into callbacks or a custom training step. Neither presentation changes the underlying objective; choose the boundary that makes the current task easiest to verify.

Layout is a practical example. PyTorch convolution layers conventionally read images as (batch, channels, height, width). The Keras implementation receives the same fixture after an explicit conversion to (batch, height, width, channels). The values and labels are shared; only their documented layout changes. Omitting this conversion could produce an error, but worse, a shape

that happens to be accepted might make a different network from the one being compared.

For a comparison to be reviewable, write a protocol before running either program. Name the question, freeze the fixture revision, list inputs and split rule, declare preprocessing/layout conversion, define topology and optimizer, state seed and budget, select metrics, and state what will not be compared. Elapsed time is recorded here as an observation but excluded from any framework-speed conclusion.

When results diverge, debug in order. Compare data counts, class names, labels, and split membership. Then inspect preprocessing and tensor layout. Next compare logits/loss conventions, optimizer settings, batch rule, and epoch count. Only then investigate initialization, kernels, device visibility, or defaults. This order keeps a mundane data error from becoming a grand framework argument.

Real implementation: compare the contract before the syntax

The comparison record at benchmarks/02-vision/README.md fixes the train, validation, and test partitions at 32, 16, and 16 examples per class. It also requires held-out accuracy, a three-by-three confusion matrix, and a list of mistakes. Those checks matter more than matching a particular layer spelling: they tell us whether each program has learned the same stated task.

Read the comparison table as a contract, not a leaderboard. The partitions are fixed before either model is trained. Accuracy is evaluated separately, the confusion matrix keeps per-class results visible, and the error list preserves the examples behind a metric. For this separable fixture, zero test errors are expected and still recorded explicitly. In a realistic vision problem, inspect representative errors, class imbalance, ambiguous labels, and changes across seeds before drawing conclusions from one score.

Fairness has limits even in this matched exercise. The PyTorch loop uses the full training fixture as one batch while Keras uses batches of 32, and the frameworks choose their own initial parameter values. These choices can affect loss trajectories and elapsed time, so they are documented rather than hidden. The educational question is whether both programs solve the declared held-out task, not whether their internal states are bit-for-bit identical.

The confusion matrix is not decoration. Rows represent actual classes and columns predicted classes in the shared helper. An identity matrix with sixteen examples on each diagonal says every held-out stroke type was predicted as itself in this fixture. The accompanying empty mistake list has meaning only because its index space, labels, and split are fixed. On a harder task, save representative mistakes and inspect whether they cluster around one class, noise level, or preprocessing transformation.

The topology is equivalent at the level that matters to this lesson, not a claim of identical parameter tensors. Both models apply same-padded 3×3 convolutions, ReLUs, pooling, global average pooling, and three final logits. Different libraries initialize weights differently and may order or fuse operations differently. Requiring bit-for-bit agreement would turn a lesson about controlled task behavior into a fragile implementation test. Requiring the declared fixture, held-out evaluation, and error report instead makes the comparison useful and reproducible.

The comparison needs a stopping rule. Once both implementations meet the declared held-out contract on this toy fixture, attempts to match every loss digit do not improve the lesson. They can overfit the experiment to details. Record remaining differences and design a new question when one matters: a custom gradient step, export path, data-pipeline constraint, deployment target, or controlled performance workload.

Experiment

On the recorded Mac, PyTorch 2.13.0 completed the fixture with MPS selected and also with CPU explicitly selected. TensorFlow/Keras 2.21.0 saw CPU as its only visible device. Each recorded run reached 1.000 train, validation, and test accuracy with an identity confusion matrix (16 held-out examples correct in each class) and no mistakes. The raw results, versions, and elapsed values are in benchmarks/02-vision/README.md.

That is task-level agreement, not a framework performance result. The runs use different device paths and only one timing observation per configuration; neither a loss value nor elapsed milliseconds establish that one framework is faster. The final losses also differ because initializers and implementation details differ. A proper speed comparison needs repeated runs, shared warmup, equivalent device selection, and a declared timing boundary.

The MPS PyTorch run is slower than the recorded CPU run for this tiny workload. That is not a contradiction of the idea that accelerators can help: setup, dispatch, and synchronization can outweigh parallel computation when the work is small. It is also not proof that CPU is generally faster. The only supported statement is the one in the record: these wall-clock values were observed under the declared, one-run conditions. Treat any broader conclusion as a hypothesis requiring a larger, repeated, device-matched experiment.

Reproducibility also has levels. Fixed fixture generation and explicit seeds make this CPU-friendly teaching task repeatable. They do not guarantee identical floating-point traces across library versions, devices, or kernels. Record the versions, selected device, data layout conversion, and observed final metrics; then treat an unexpected divergence as an investigation prompt. A seed narrows uncertainty; it is not a promise that every numerical detail is portable.

Device visibility is a difference that should remain in the record rather than be normalized away. The recorded TensorFlow install exposed CPU only, while the PyTorch auto run selected MPS. That prevents an accelerator comparison but not the correctness comparison. A fair speed protocol needs matching devices or a scoped CPU-only condition, repeated warmups, matching batch semantics, and a defined timing boundary.

Worked comparison: agreement on the task, not on every number

Start with the four invariant checks from the record. Both programs receive 16×16 grayscale strokes from the same deterministic generator; each class has 32 training, 16 validation, and 16 test examples; the labels remain vertical, horizontal, and diagonal; and both report held-out accuracy, a confusion matrix, and mistakes. The one necessary layout conversion is visible: the shared NCHW fixture becomes NHWC for Keras. A reviewer can therefore test whether the learning question stayed the same before reading either training API.

The recorded result is a three-by-three identity confusion matrix with sixteen examples in every diagonal cell for all three runs. PyTorch MPS, PyTorch CPU, and TensorFlow/Keras CPU each report 1.000 train, validation, and test accuracy with an empty mistake list. This is task-level agreement on a deliberately separable synthetic fixture. It does not show that the

frameworks will agree on natural images, an imbalanced dataset, another random seed, or a more difficult split.

Now compare the fields that should *not* be collapsed. Final train loss is 0.275691 for the recorded PyTorch MPS run and 0.001079 for TensorFlow/Keras. Their parameter initialization and implementation details are not identical, so the loss values are not a winner/loser score. Elapsed time is also scoped: the MPS, PyTorch CPU, and TensorFlow CPU observations are 2323.301 ms, 404.449 ms, and 1015.773 ms respectively, each from a single declared run with unequal device and batching conditions. Those values are an environment trace, not a speed table.

If a new Keras run produced an off-diagonal error, investigate the contract in order. Confirm that to_nhwc preserved image values and labels; confirm the fixture split/counts and fixed seed; then inspect sparse-label loss/logit conventions, batch behavior, and epoch budget. Only after those match should you inspect initializer or kernel differences. This sequence turns a surprising metric into a localized experiment question instead of a claim that one framework's philosophy is better.

What broke

Framework setup was the first practical difference. The core project does not install TensorFlow, so the Keras experiment requires uv sync --group tensorflow. On this Mac it ran on CPU only and emitted a non-fatal use_unbounded_threadpool NodeDef compatibility warning; training and evaluation still completed. Treat such a warning as an environment observation to investigate, not as evidence that the result is invalid or portable.

A more subtle failure is false equivalence. Matching names such as Conv2d and Conv2D is insufficient if layouts, padding, shuffling, splits, or metrics change. The shared fixture converts PyTorch's channels-first tensor to Keras's channels-last input explicitly, then verifies the same held-out labels.

Data leakage is another comparison failure. If generated images from the same random pattern, augmentation lineage, or preprocessing cache cross the split boundary, both frameworks may report excellent held-out accuracy while the evaluation question has been compromised. The fixture uses distinct declared seeds for its splits. When replacing it with a real dataset,

preserve the split manifest and fit normalization statistics only on the training partition.

Configuration drift is another failure mode. A helper's default shuffling, loss reduction, data format, or seed behavior can change across an upgrade. Capture resolved versions and print runtime choices with each run. A lockfile makes installation repeatable; it does not prove unprinted defaults stayed equivalent.

Alternatives and when to use them

Use PyTorch when an explicit training loop and immediate tensor inspection help you learn or debug. Use Keras when a compact model-and-training declaration is a better fit for the team and deployment path. JAX, higher-level PyTorch trainers, and other ecosystems are valid alternatives, but should enter only after the evaluation contract is stable.

Use a custom Keras GradientTape loop when the experiment requires the same step-level control demonstrated in PyTorch; use PyTorch's module and optimizer interfaces when an explicit loop remains clearest. Avoid choosing from benchmark headlines when hardware, model size, precision, data pipeline, or deployment target differs from the work at hand. A repeatable question and a maintained test fixture are more durable assets than a short-lived ranking.

For a production comparison, expand the fixture before expanding the rhetoric. Use a versioned real-data manifest or a more varied synthetic generator, preserve train-only preprocessing, select a metric that reflects actual error cost, and repeat across declared seeds. Only after correctness is fixed should a separate performance protocol select matching devices, warm-ups, precision, batch sizes, run counts, and timers. The two reports then answer different questions without smuggling one conclusion into the other.

The same protocol applies beyond these two libraries. A JAX implementation, high-level PyTorch trainer, or exported inference runtime should inherit the shared dataset and evaluation harness before it earns a comparison table. The goal is not identical programs; it is localized differences a reader can judge.

Evidence trail

The shared framework note is research/02-vision-and-frameworks/notes.md. Run experiments/05-vision/train_pytorch_cnn.py and experiments/04-tensorflow/train_keras_cnn.py; compare their recorded contract in benchmarks/02-vision/README.md, not elapsed values alone.

Takeaway

Framework choice is an engineering trade-off, not a different theory of learning. Start with a shared question and evidence contract; then let API clarity, operational requirements, and measured constraints determine the tool.

Run the lab on your Mac

QR code for Chapter 04 lab

Online lab: https://hassanvfx.github.io/mac-ai/labs/04

Repository: experiments/04-tensorflow/train_keras_cnn.py

```
uv run --group tensorflow python experiments/04-tensorflow/train_keras_cnn.py --epochs 30
```

Expected: Matched deterministic CNN metrics.

Evidence: benchmarks/02-vision/README.md

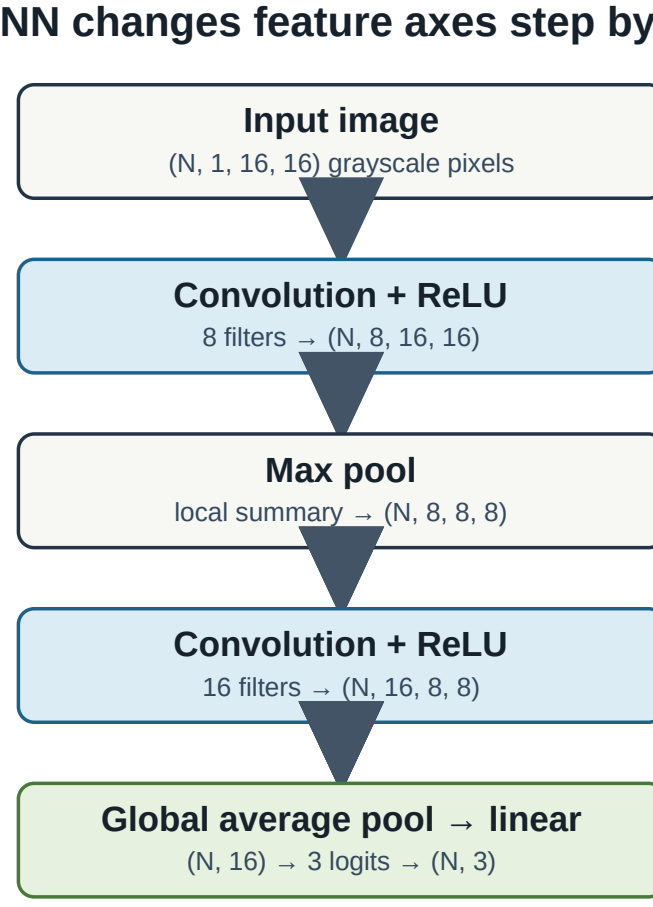
Scan the code to open the live lab, then use Safari Share → AirDrop to send it to your Mac. The lab follows main; clone the repository once and run the command from its root directory.

How Neural Networks Learn to See

Intuition

Convolutions use local structure and repeated filters to turn pixels into useful features. A filter that responds to a short vertical stroke can be useful at many image positions, so a convolution shares the same learned weights across the image. Pooling and later layers combine local responses into a decision about the whole image (Goodfellow et al. 2016).

The important idea is *inductive bias*: a convolution assumes that nearby pixels matter together and that a useful local pattern may appear in more than one position. A fully connected layer can in principle learn such patterns, but it assigns a separate weight to every input connection. A convolution uses one small kernel repeatedly, which reduces parameters and expresses the useful prior that a vertical edge is still a vertical edge when it shifts a few pixels.



Convolution changes channels; pooling reduces spatial detail.

The tiny CNN changes channels and spatial resolution while preserving the batch.

The receptive field grows as layers compose. A first 3×3 filter sees a small patch; after pooling and a second convolution, a feature can depend on a wider region of the original 16×16 image. Global average pooling then summarizes each feature map across all remaining positions. This does not mean the network has learned human concepts such as “line” or “object.” It means the architecture offers a route from local pixel patterns to a translation-tolerant class score.

The phrase “feature map” names an intermediate tensor, not a verified picture of what the network sees. One channel may respond strongly to a stroke-like pattern in this fixture, but its values are the result of learned filters, nonlinearities, and the data distribution—not a human-approved concept label. Visualizing feature maps can help debug dimensions, dead channels, or saturated activations. It cannot prove that a prediction was caused by a visual story a reader prefers.

Receptive field has a boundary. A wider field lets a later unit combine more of the image, but it does not restore detail a pooling step or crop removed. For this 16×16 fixture, two convolutions and pooling make a compact teaching model. For a high-resolution task, the same choices can discard thin objects or exact positions. Begin with “what must remain distinguishable?” rather than layer count.

Problem

Train and inspect a small image classifier rather than treating a CNN as a black box. We need more than a final accuracy: the split must be held out, the output classes must be named, and errors must be inspectable.

Minimal implementation

The companion fixture creates three balanced classes of 16×16 grayscale images: a vertical, horizontal, or diagonal noisy stroke. It is generated from fixed seeds and stored as code, so a reader does not depend on an external download or an unstable train/test split.

Run the complete PyTorch implementation:

```
uv run python experiments/05-vision/train_pytorch_cnn.py --device auto --epochs 30
```

The model is deliberately small:

```
image → Conv(8) → ReLU → max pool → Conv(16) → ReLU → global average pool
→ 3 logits
```

Logits are unnormalized class scores. Cross-entropy compares them to the known class index, and Adam changes the filters to reduce that loss.

The actual tensor path is worth writing down. The fixture produces a channels-first batch (N, 1, 16, 16). The first padded 3×3 convolution keeps the

16×16 spatial grid while changing one input channel into eight learned feature maps. Max pooling halves height and width to (N, 8, 8, 8). The next convolution creates 16 feature maps, and adaptive global-average pooling turns each entire map into one number. Flattening produces (N, 16), then the final linear layer emits (N, 3) logits—one score for each named class.

Pooling does not discover an object; it discards some exact location detail in exchange for a compact local summary. On this fixture, that is a reasonable choice because a stroke may be offset slightly. On a task where exact position is the answer, aggressive pooling can remove the signal we need. Architecture is therefore a hypothesis about the task, not a collection of default layers.

Real implementation: an inspection-friendly evaluation

src/from_tensors_to_agents/vision.py owns the fixture, model, class names, and confusion-matrix function. The training script reports train, validation, and test accuracy separately; it also prints the confusion matrix with actual classes in rows and predicted classes in columns, followed by every held-out mistake. This output is intentionally simple enough to test in CPU CI.

Separate modes make evaluation meaningful. During training, model.train() declares that train-time behavior should be active; before producing test predictions, the script uses model.eval() and torch.no_grad(). The tiny network has no dropout or batch normalization, so the numerical effect is small here, but preserving the boundary establishes a habit needed for larger models. no_grad() also says that prediction is not part of a later update, avoiding unnecessary graph storage.

The confusion matrix answers a question accuracy hides: *which* cases are wrong? Rows are actual classes and columns are predicted classes. If diagonal strokes were repeatedly predicted as vertical, the corresponding off-diagonal cell would identify a particular failure direction. The mistake list supplies individual fixture indexes to inspect. A zero-error matrix is not empty evidence; it documents that none of the declared error categories occurred on the frozen held-out data.

The loss and metric answer different questions. Cross-entropy uses every logit and known target to create a differentiable learning signal; accuracy discards margin information and counts only whether the highest logit has

the right class. A model can improve cross-entropy while accuracy stays unchanged, or reach the same accuracy with more or less decisive scores. Keep both when debugging training, then inspect examples and the confusion matrix before deciding whether either result is useful.

Validation data guides choices such as epoch budget, architecture, or noise level. Test data estimates the final declared task after those choices are made. Repeatedly using the test score to select a configuration turns it into another validation set. This small fixture keeps all three splits because the habit is more important than the difficulty of the current images.

A decision rule for the three splits

Use training data to compute gradients. Use validation data to compare a small number of predeclared alternatives: two learning rates, a pooling choice, or a noise condition. Freeze the selected configuration, then evaluate the test split once as the reported result. If a test result prompts another architecture change, the old test set has become model selection and a new held-out split is needed for an honest final estimate.

On a small fixture, record counts as well as percentages. Accuracy of 1.000 on 48 held-out examples means 48 declared predictions were correct; it does not estimate a large-population error rate precisely. The identity matrix and empty mistake list add detail, but do not make the sample larger.

From an error to a hypothesis

If a harder follow-up makes diagonal strokes become vertical predictions, do not immediately add layers. Open the named examples, compare their conditions with correct diagonals, and check preprocessing and labels. Then state a falsifiable hypothesis—perhaps pooling removes a displaced diagonal cue—and test one intervention against the same fixed test condition. This connects an off-diagonal cell to an experiment rather than an anecdote.

Experiment

The recorded PyTorch MPS run used 32 training examples per class, 16 validation examples per class, 16 held-out test examples per class, seed 17, Adam at 0.01, and 30 epochs. It reached 1.000 accuracy on each split. Its held-out matrix was the identity matrix: each of the 16 vertical, 16 horizontal, and 16 diagonal test examples was classified correctly. The explicit CPU run

produced the same accuracy and zero mistakes. See benchmarks/02-vision/README.md for the complete record and timing limitations.

Zero mistakes are still an error-analysis result: there were no misclassified examples to inspect under this controlled distribution. They do *not* mean the network can see in the everyday sense. The images are highly separable, the same generator supplies every split, and the test set is small. A next experiment should make the boundary less clean—for example by varying stroke position, contrast, or noise—and report which classes then confuse one another before considering a real image dataset.

Use this result as a baseline, then make one change at a time. Increase noise without changing split sizes and inspect whether errors first concentrate in diagonal examples. Translate strokes farther from the center and ask whether pooling helps enough. Reduce training examples per class and track the gap between train and validation accuracy. Each variation tests a concrete hypothesis about data, rather than asking a larger model to solve an undefined problem. Commit the changed generator settings and result before describing a robustness finding in prose.

To create a controlled-difficulty follow-up, change exactly one generator condition: raise Gaussian noise, increase translation range, thin strokes, or reduce training examples. Keep class balance, test seed, topology, and metrics fixed. State the hypothesis before running—for example, "diagonal strokes will be confused with vertical strokes under larger offsets"—then inspect the matrix and saved mistakes. A harder score alone is not a finding until its error shape and changed condition are recorded.

Keep a baseline record immutable. A controlled variant needs its own generator settings, split seeds, model/optimizer settings, device, metrics, matrix, mistakes, and limitations. Otherwise a later noisy run can overwrite the perfect baseline and make it impossible to identify what changed. Benchmark records preserve this history; prose should point to the executable condition.

Worked error-analysis case: accuracy needs a direction

The current baseline has no held-out mistakes, so use the small confusion-matrix unit test to practice reading a non-perfect result. Its actual labels are [vertical, horizontal, horizontal], while its predictions are [vertical, diagonal,

horizontal]. With actual classes as rows and predicted classes as columns, the matrix is:

```
             predicted
actual     vertical  horizontal  diagonal
vertical          1           0         0
horizontal        0           1         1
diagonal          0           0         0
```

The single off-diagonal cell says more than "two out of three were correct." One horizontal example was predicted as diagonal. It does not say that every horizontal image is hard, that diagonal filters are wrong, or that the network needs another layer. It identifies a direction for inspection: open the named fixture index, compare its stroke position, thickness, and noise with correct horizontal examples, and verify the label and preprocessing before changing an architecture.

Turn that observation into a narrow hypothesis. For example: "larger vertical offsets make horizontal strokes resemble diagonals after pooling." Create a new fixture configuration that changes only the offset range, preserve the class balance and split seeds, and run the same train/validation/test protocol. If the horizontal-to-diagonal cell grows while other conditions stay fixed, the result supports a specific robustness question. If it does not, the original mistake may have been an isolated noisy example rather than evidence for a design change.

The baseline's identity matrix remains valuable beside this hypothetical case. It establishes that the declared easy distribution has no errors; the variant would establish how one changed distribution behaves. Do not overwrite the identity matrix or report only the worse aggregate accuracy. Versioned matrices and mistake lists let a future reader distinguish a changed task from a regression in the original one.

What broke

The easy failure is to report training accuracy alone. A model can memorize a tiny fixture while performing poorly on unseen data, so this experiment keeps validation and test partitions separate. Another easy failure is a layout mismatch: PyTorch convolution expects channels-first inputs while Keras uses a channels-last version of the same fixture. The matched framework experiment converts that layout explicitly rather than relying on an implicit transpose.

Finally, do not call this a general CNN benchmark. Its purpose is to make the flow from pixels to logits inspectable and deterministic; it does not measure augmentation, robustness, scale, or real-world generalization.

An incomplete error analysis is a failure too. A single accuracy value omits whether one class was never recognized, whether errors repeat a visual pattern, and whether mistakes came from training, validation, or test data. Keep class names and matrix ordering with the recorded result. When a real dataset replaces this fixture, examine labels as well as predictions: a model cannot learn a distinction that annotators apply inconsistently.

Distribution shift is the larger version of the same problem. The fixture assumes centered-ish strokes, clipped Gaussian noise, and three balanced named classes. A camera image with texture, a new diagonal angle, or an unrepresented class asks a different question. A confident output is still only one of the original three logits. Before using a CNN in a new environment, collect representative held-out data, decide how unknowns are handled, and evaluate the error patterns that matter to users.

Alternatives and when to use them

For image-like grids with local patterns, convolution gives a useful inductive bias and often needs less data than a fully connected network. A small linear classifier is a worthwhile baseline when the relationship may be simple. For large-scale vision work, residual CNNs and vision transformers are common alternatives, but they add capacity and evaluation requirements that would hide this chapter's basic lesson.

Data augmentation is an alternative when the real deployment distribution contains harmless rotations, crops, or illumination changes. Apply it only to the training partition and state exactly which transformations are permitted; augmenting held-out examples invalidates the evaluation boundary. Residual CNNs can train deeper feature extractors, and vision transformers use patch tokens and attention instead of convolution. Both are useful later, but neither makes a tiny, perfectly separable fixture a proxy for real-world vision.

There is also a lesson in the fixture's construction. Train, validation, and test images use separate random seeds, but share the same generating family. That is appropriate for a controlled lesson about the learning loop, while also limiting what the result establishes. A real deployment may have camera noise, backgrounds, labels, and class proportions that this generator never

models. The correct response is not to claim robustness from a toy score; it is to create a new versioned evaluation question with those conditions represented.

Segmentation, detection, and keypoint tasks are further alternatives when a single class label discards information the product needs. Their outputs carry spatial structure, so datasets, losses, metrics, and error review must do the same. A classifier that calls a scene "diagonal" cannot replace a system that must identify where a defect lies. Match output representation to the decision the user must make before choosing an architecture family.

Evidence trail

The fixture and CNN are defined in src/from_tensors_to_agents/vision.py; the research note is research/02-vision-and-frameworks/notes.md. Run experiments/05-vision/train_pytorch_cnn.py and inspect the evidence record in benchmarks/02-vision/README.md.

Takeaway

Neural networks learn to see by turning repeated local measurements into a loss-driven decision. Trust the result only as far as its held-out data, confusion matrix, error list, and stated limitations allow.

Run the lab on your Mac

QR code for Chapter 05 lab

Online lab: https://hassanvfx.github.io/mac-ai/labs/05

Repository: experiments/05-vision/train_pytorch_cnn.py

```
uv run python experiments/05-vision/train_pytorch_cnn.py --device auto --epochs 30
```

Expected: Train, validation, and test metrics with errors.

Evidence: benchmarks/02-vision/README.md

Scan the code to open the live lab, then use Safari Share → AirDrop to send it to your Mac. The lab follows main; clone the repository once and run the command from its root directory.

The Transformer Changed Everything

Intuition

Attention lets each token select useful context rather than receiving a fixed summary of a sequence. Self-attention compares each token with other tokens in the same sequence, then mixes their value representations according to those comparisons. Stacking that operation lets a Transformer form context-dependent representations without a recurrent step-by-step state (Vaswani et al. 2017).

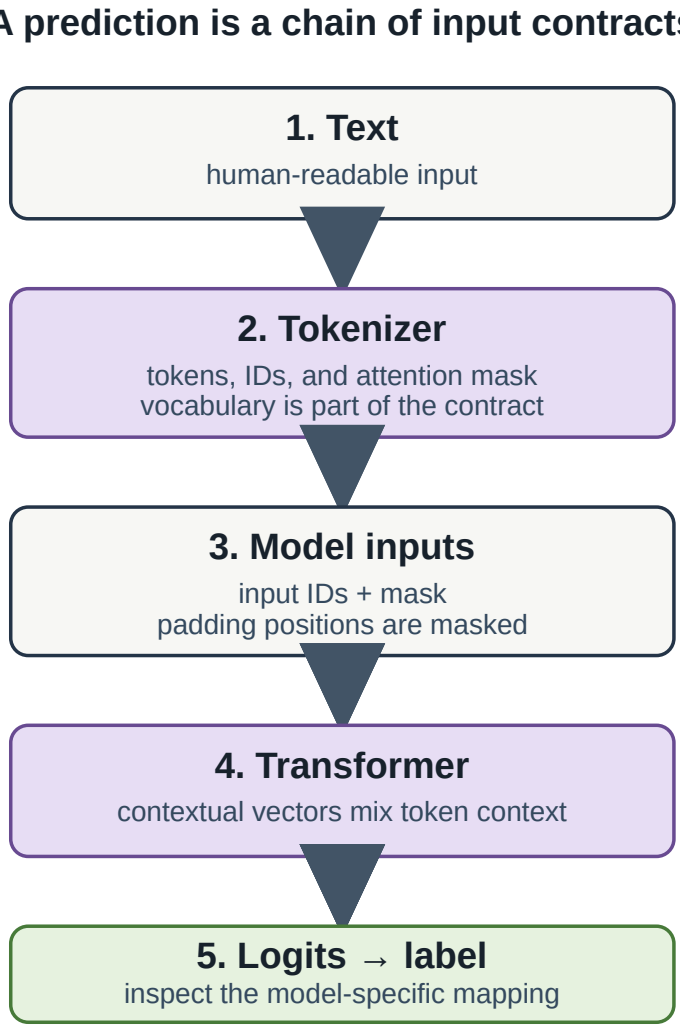
That description can sound abstract until we inspect the actual inputs. A pretrained Transformer does not receive words directly. It receives token IDs, special boundary markers, and an attention mask; it returns scores whose meaning comes from the model's trained label mapping.

Tokenization is a learned system boundary, not a cosmetic preprocessing step. The tokenizer chooses a vocabulary and a rule for decomposing text into entries that model was trained to recognize. WordPiece may retain a common word as one token while splitting an unfamiliar word into smaller pieces. The resulting integer IDs have no inherent numeric order: ID 200 is not more positive or more important than ID 20. They are lookup keys for learned embedding vectors.

Self-attention then lets a token representation depend on other positions in the same input. Each layer forms query, key, and value projections; similarity between a query and keys becomes a set of weights, and the weights mix values. Multiple attention heads can specialize in different relationships. This is not a built-in explanation of why a model chose a label: attention weights are part of the computation, while a trustworthy explanation requires a separate, validated evaluation question.

At one position, attention can be written compactly as $\text{softmax}(QK^\top / \sqrt{d_k})V$. The formula names an operation, not an interpretation: queries and keys create compatibility scores, scaling keeps their magnitude manageable, softmax normalizes them across permitted positions, and values are mixed using the resulting weights. Residual connections, feed-forward layers, normalization, many heads, and many stacked blocks make the full model

more than this one line. Track tensor contracts and input masks before trying to tell a story about what any head "means."



Transformer inference turns text into an explicit sequence of model contracts.

Problem

Move from tokenization to a pretrained model inference path we can inspect. We will classify two fixed sentences with a compact binary sentiment model. This is a demonstration of the mechanics of pretrained inference, not proof that a model understands sentiment, evaluates quality, or makes a trustworthy recommendation.

Minimal implementation

Install the optional dependency group and run the complete inspection:

```
uv sync --group transformers
uv run --group transformers python experiments/06-transformers/inspect_sentiment.py
--device auto
```

The script loads distilbert/distilbert-base-uncased-finetuned-sst-2-english at its recorded revision. For the sentence This small experiment is clear and useful., its WordPiece tokenizer emits:

```
[CLS] this small experiment is clear and useful . [SEP]
```

The corresponding integer IDs and all-ones attention mask are printed by the script. [CLS] and [SEP] delimit the sequence for this model family; they are not words from the sentence. An attention mask marks which positions are real input rather than padding when a batch contains sequences of unequal length.

Padding illustrates why masks exist. A batch processor wants rectangular tensors, but one sentence may have 10 tokens and another 40. It can add padding tokens to make both rows length 40. The attention mask distinguishes genuine positions from added placeholders, so the model does not treat padding as evidence. Inspect the mask whenever batching or truncation behaves oddly: an incorrect mask can produce a numerically valid forward pass with the wrong effective input.

A worked masked-attention view

Imagine a batch with one sequence [CLS], good, [SEP] and one shorter sequence [CLS], bad, [SEP] padded to the same width. Their masks might both be [1, 1, 1] in this tiny example; with extra padding, the shorter row becomes [1, 1, 1, 0, 0]. Before softmax, an attention implementation adds a

very negative value to scores for masked key positions. Softmax then assigns those positions effectively zero weight. The mask does not give the model a new fact; it prevents an artificial padding symbol from becoming a candidate source of context.

This is a contract on both axes. The mask's batch and sequence dimensions must refer to the same examples and positions as the token IDs. A transposed or misaligned mask can have the correct numeric shape in a square batch while silencing real tokens or exposing padding. When attention output looks odd, inspect decoded tokens and masks together before inspecting model weights.

Softmax also makes attention weights relative. If one score rises while all others remain fixed, its normalized share can rise even though its raw value is not a general measure of importance. A token can receive high attention in one head/layer while later residual paths and feed-forward layers change the final representation. Use attention tensors to test a computation or debug a mask, not as a standalone explanation for a classifier decision.

Real implementation: reveal the path hidden by a convenience API

The manual path in experiments/06-transformers/inspect_sentiment.py is:

```
text → tokenizer → input IDs + attention mask → model logits → softmax → labels
```

It uses AutoTokenizer and AutoModelForSequenceClassification, moves tensors and the model to MPS when available (otherwise CPU), runs under torch.inference_mode(), and applies softmax explicitly. The high-level pipeline("text-classification") path then uses the same model, tokenizer, device, inputs, and maximum length. Transformers provides both styles because one optimizes inspection and the other reduces application boilerplate (Wolf et al. 2020; Hugging Face 2026).

Logits are the model's raw, unnormalized class scores. Softmax converts two or more logits into non-negative values that sum to one; for a fixed label set, the largest softmax value selects the predicted label. A high score is not a probability that the sentence is objectively positive, nor a measure of how well the model will work on another domain. It is confidence within this model, revision, label mapping, and input representation. Preserve all four when recording an inference observation.

The direct path is the right debugging baseline because it makes every transformation visible. Print the decoded tokens, IDs, mask, logits, sorted labels, and selected device before trusting an application wrapper. Once that path is known to be equivalent on fixed inputs, a pipeline is useful for small applications and demonstrations. If it later disagrees, reduce back to the direct path first rather than guessing which convenience default changed.

The two paths should match only under a shared contract. Keep model revision, tokenizer revision, label mapping, device, truncation setting, maximum length, and batch/input order fixed. A pipeline may otherwise choose defaults that are reasonable for an application but unsuitable for an investigation. In this example, the manual path runs first so decoded tokens and ranked logits become the reference for interpreting the pipeline output.

Softmax values sum to one across classifier labels for this input; they do not tell us whether the label set is suitable, whether training data resembles the input, or whether a sentence has one objective sentiment. Calibration is a separate empirical property. When a product decision depends on confidence, evaluate calibration and error costs on a relevant held-out dataset instead of thresholding a displayed score by intuition.

What an inference evaluation would need

The two fixed sentences are an interface fixture, not an evaluation dataset. They answer whether the manual and pipeline paths agree under a declared model revision. An inference evaluation needs a versioned collection of inputs with labels or human judgments, a sampling rule, a metric appropriate to the task, and a plan for failures. For sentiment classification that may include class balance, accuracy or F1, calibration, domain slices, negation/sarcasm cases, and a human review rule for uncertain or harmful errors.

Keep model evaluation separate from prompt anecdotes. A prompt that produces an intuitive label is a useful demonstration, but it is selected evidence. A good test set includes cases the author did not choose because they flatter the model. Record model identifier, revision, tokenizer, maximum length, preprocessing, device, and label mapping with the results so a future reader can tell whether a changed score came from the model or the evaluation setup.

Experiment

The recorded run used Python 3.11.9, PyTorch 2.13.0, Transformers 5.15.0, a maximum sequence length of 64, and model revision 714eb0fa89d2f80546fda750413ed43d93601a13. On MPS, the first sentence was classified POSITIVE at 0.999802 and the second, explicitly unfavorable sentence was classified NEGATIVE at 0.999689. The direct path and pipeline matched in top label and score for both fixed inputs. CPU produced the same two results. See benchmarks/03-transformers/README.md for the full record.

The direct two-sentence pass took 380.206 ms in the recorded MPS run and 120.787 ms on the recorded CPU run. This surprising-looking pair is not a speed comparison: it is a single tiny workload without repeated warmups or equal overhead boundaries. It is included precisely to demonstrate why a number without a benchmark design should not become a hardware claim.

Reproduce the interface check before changing the workload. First inspect the JSON record: tokens, IDs, masks, model revision, device, logits-derived ranking, and pipeline result. Then change one variable—such as --max-length or --device cpu—and keep the original record. If the label changes, locate the first changed contract: tokenization, truncation, model revision, label map, or device. A changed label is an observation that needs context, not a reason to select the more convenient result.

For a context-length policy, define what happens to text that exceeds the model's input window before deployment. Truncating the tail may be acceptable for a short classification field whose relevant information appears first; it is unsafe for a document where the conclusion or exception appears last. Chunking produces multiple predictions that need a declared aggregation rule; summarizing before classification introduces a second model and failure mode. Choose the policy from the task and evaluate boundary cases, not from whichever option makes an API call easiest.

Worked audit: from tokens to one ranked label

Take the favorable fixture sentence and begin with the printed token sequence: [CLS] this small experiment is clear and useful . [SEP]. The special boundary tokens and all-ones mask are part of the model input, while the displayed words are only a decoded aid for the reader. The token IDs are opaque vocabulary indexes; do not compare their magnitude or treat them as sentiment features. The first audit question is simply whether the tokenizer,

mask, maximum length, and selected model revision describe the input you intended to classify.

The manual path then produces two logits, applies softmax, and ranks labels using the loaded model's id2label mapping. In the recorded MPS observation, the top label is POSITIVE at 0.999802 for this sentence. The companion pipeline uses the same model, tokenizer, truncation setting, and input; its top label and score match the manual path. The test helper enforces the small structural version of this rule: when manual logits rank POSITIVE first and a pipeline result reports the same score, the comparison records equal labels and zero absolute score difference.

This agreement is an interface check, not a truth check. It shows that two routes through one declared artifact present the same top result for a selected input. It does not show that 0.999802 is calibrated on another domain, that sentiment is an objective property of the sentence, or that a high-attention token caused the label. The unfavorable fixture sentence supplies a second control input; an evaluation would need a versioned set of many unselected examples and a decision rule for mistakes.

To diagnose a disagreement, work backwards from the output. Confirm the label mapping before assuming the label name; compare tokenizer IDs and mask; verify the model/tokenizer revision pair; then compare truncation, maximum length, batch order, and device. A different top label after changing --max-length may be a context-policy observation, not a pipeline bug. Preserve both JSON records so the first changed contract remains visible.

What broke

The first practical surprise is storage. Loading a pretrained model downloads and caches files outside the repository; this particular cache occupied about 256 MiB after the run, but model sizes vary enormously. Check disk space before selecting a model and never commit the cache.

The second surprise is that a label is not an explanation. The fixed favorable and unfavorable sentences were selected to make the control flow easy to read, not to evaluate the model. Inputs near a decision boundary, different domains, long contexts, sarcasm, and unfamiliar language can behave very differently.

Finally, context length is an input contract. The experiment enables truncation at 64 tokens. In a real system, silently discarding the tail of a document can change the result; record the limit, inspect the tokenized sequence, and choose a chunking strategy rather than assuming the model saw everything.

The 64-token limit is especially easy to misuse in document work. Token count is not word count, and a boundary can split an important qualifier from the claim it limits. For longer material, decide whether to shorten, chunk, pool, or use a model with an appropriate context window. Record the policy and test examples near the limit. Silent truncation can make a result look stable while excluding the evidence a reader cares about.

Labels also drift across checkpoints. Do not hard-code "class 1 means positive" without reading the loaded model's id2label mapping. A different fine-tuned checkpoint can reverse a mapping, add classes, or use labels whose names are less self-explanatory. The manual ranking helper reads the mapping to keep this experiment tied to the selected model artifact.

Tokenizer/model mismatch is a related failure. Token IDs are meaningful only to the embedding table and vocabulary that trained with them. Loading a tokenizer from a different checkpoint, silently changing special-token handling, or mixing cased and uncased preprocessing can preserve a valid tensor shape while altering the input the classifier actually receives. Pin the paired model and tokenizer revision for any recorded inference result.

Alternatives and when to use them

Use a direct model call when you need to inspect preprocessing, logits, labels, or device placement. Use a pipeline for a well-understood inference task after the low-level path has been verified. Recurrent models remain useful where streaming state or very small deployments dominate; task-specific classifiers can be simpler and more predictable than a general generative model.

For offline inspection, a small local classifier is useful because its inputs and output space are bounded. For open-ended synthesis, an instruction model has different context, safety, and evaluation requirements; do not treat this sentiment script as a miniature chat assistant. For exact strings, metadata, or known keywords, conventional search and rules can be more transparent than any Transformer inference call.

Encoder classifiers, decoder-only language models, and encoder-decoder models also solve different interface problems. A compact encoder classifier maps a bounded input to a known label set; a decoder predicts the next token repeatedly; an encoder-decoder model maps one sequence to another. The word "Transformer" does not make their context policies, outputs, evaluation methods, or safety requirements interchangeable. Choose the architecture after naming the input, output, and error decision the product actually needs.

Evidence trail

Read research/03-transformers/notes.md, run experiments/06-transformers/inspect_sentiment.py, and use benchmarks/03-transformers/README.md for the fixed-input observation and its timing limits.

Takeaway

Transformers trade recurrent sequence processing for flexible context mixing, but they still execute a precise input-to-output contract. Inspect the tokens, mask, logits, label mapping, model revision, device, and limitations before turning a convenient prediction into an engineering conclusion.

Run the lab on your Mac

QR code for Chapter 06 lab

Online lab: https://hassanvfx.github.io/mac-ai/labs/06

Repository: experiments/06-transformers/inspect_sentiment.py

```
uv run --group transformers python experiments/06-transformers/inspect_sentiment.py
--device auto
```

Expected: Tokenizer and inference inspection.

Evidence: benchmarks/03-transformers/README.md

Scan the code to open the live lab, then use Safari Share → AirDrop to send it to your Mac. The lab follows main; clone the repository once and run the command from its root directory.

Your Mac Is an AI Workstation

Intuition

Apple Silicon uses unified memory: the CPU and GPU-accessible parts of a system share a memory pool rather than copying every model tensor through a separate GPU memory space. That can make local inference practical, but it does not make memory infinite or a model automatically fast. Model size, quantization, prompt length, generated tokens, allocator behavior, and other applications all still matter.

Unified memory changes a systems boundary, not the laws of capacity. A local model, its tokenizer, generated-token cache, application process, browser, and the operating system all compete for one finite pool. Quantization stores some model weights with fewer bits, but it does not remove every runtime allocation or make a model's context free. The practical question is therefore not whether an advertised parameter count sounds small, but whether this exact model and workload leave enough headroom for the rest of the machine.

Storage and memory are separate constraints. The first model download needs disk space for weights and tokenizer files; a later generation needs available unified memory for the loaded model, runtime buffers, prompt, key/value cache, and everything else running on the Mac. Checking df -h . before an optional download prevents a simple disk failure, but it says nothing about whether a long prompt will fit comfortably at inference time.

Problem

Run one local MLX-LM inference workload whose inputs and measurement boundary are visible. A fair PyTorch-MPS-versus-MLX comparison comes later, after the same generative model and protocol are available in both runtimes.

Before running it, turn the vague request "try a local model" into an experiment envelope. Name the model artifact, the exact prompt, the maximum new tokens, decoding settings, whether the weights are already cached, the warm-up policy, and the clock boundary. Those choices determine what a result can mean. A 64-token, temperature-zero completion is useful for a reproducible smoke test; it cannot establish the latency of a

conversational application with varied prompts, streaming UI work, and longer contexts.

The envelope also gives a safe escalation path. Start with a small public model and a short prompt. Verify that the artifact downloads into a cache, the chat template renders, the response has the expected termination reason, and the process remains healthy. Only then change one variable at a time: output limit, prompt length, model artifact, or concurrent workload. A result that fails after a change is evidence about that changed workload, not proof that local inference is generally impossible.

That restraint prevents an invalid comparison. Chapter 3 measures a tiny PyTorch training loop; Chapter 6 runs a classifier; this chapter streams a quantized instruction model. Their elapsed times represent different tasks, token counts, warm-up behavior, and memory methods. A "fastest runtime" table would look precise while answering no coherent engineering question.

Minimal implementation

Install the Apple-Silicon-only optional group and run the complete example:

```
uv sync --group mlx
uv run --group mlx python experiments/07-mlx/run_local_model.py
```

The initial experiment uses mlx-community/Qwen2.5-0.5B-Instruct-4bit: a local instruction model whose identifier declares 4-bit quantization. It sends one fixed prompt through the model's chat template, performs an unmeasured four-token warm-up, and measures one deterministic (temperature=0) streamed generation capped at 64 tokens. The MLX-LM package provides model loading and generation on Apple Silicon (MLX Contributors 2026).

Read the model identifier as experiment configuration. Qwen2.5-0.5B names a particular model family and approximate parameter scale; Instruct signals a chat-oriented variant; 4bit describes the published quantized artifact. None of those words guarantees factual output, a particular memory footprint, or compatibility with another runtime. Record the complete identifier rather than shortening it to a family name, because a later revision or quantization can materially change the workload.

Real implementation: state the measurement boundary

experiments/07-mlx/run_local_model.py records model/package versions, model identifier, prompt, rendered prompt-token count, token limit, temperature, warm-up, load time, generation time, and token rates. It also records process RSS and two MLX memory views: Metal active/peak memory in bytes and MLX-LM's reported peak memory in GiB. The output labels the units rather than adding them together, because they are different partial observations.

The measurement boundary matters as much as the number. Download and model load are user-visible costs for an application, but they answer a different question from warmed-up generation throughput. The script records them separately. It also resets MLX peak memory after the warm-up so the reported generation peak does not accidentally include first-use initialization. That is not a universal memory accounting method; it is a declared observation method that a reader can reproduce and improve.

Interpret the fields as a timeline. Model load is the cold-start cost after weights are locally cached. The warm-up triggers lazy setup but is excluded from measured generation. The generation timer begins immediately before streaming and ends after the final response. If an application cares about a first interactive response, record first-token latency separately; if it cares about batch throughput, declare batch size and queueing. Do not reuse this one token rate for either question.

temperature=0 is a useful control here because the script asks for a stable decoding policy, not because it makes every layer of a model stack timeless. Tokenizers, model revisions, package versions, prompt templates, and stop conditions can still change a generated sequence. The record therefore stores versions and the rendered prompt-token count. When repeating a benchmark after an upgrade, compare the full record first; a changed output length or prompt rendering means the two runs are no longer the same workload.

For a decision rather than a demonstration, collect a small series of runs. Separate a first cached load from repeated warmed-up generations; keep the prompt and output cap fixed; retain each raw record; and report the range or distribution instead of selecting the most favorable result. If the goal is capacity planning, repeat the series while increasing context length and note

the first pressure, error, or unacceptable latency observation. Do not convert one successful short completion into a claimed maximum context length.

The memory fields are observations with different owners. Process RSS includes what the operating system attributes to Python; Metal active and peak fields describe MLX allocator views; MLX-LM returns its own peak figure. They may move together without being additive. For a longer-context experiment, preserve all three labels and record a pressure event rather than estimate capacity from one successful run.

Experiment

On the recorded Mac, MLX 0.31.2 and MLX-LM 0.31.3 generated 64 tokens after a 43-token rendered prompt in 290.701 ms, reporting 342.260 generated tokens/s. The model load took 542.927 ms and was deliberately excluded from the generation timing. MLX Metal reported 281,784,072 active bytes and 345,934,496 peak bytes; MLX-LM reported 0.345934 496 GiB peak memory. The model's local cache occupied approximately 276 MiB after download. The full contract is recorded at benchmarks/04-mlx/README.md.

These are observations for one exact model, prompt, token limit, package versions, and warmed-up run. They are not a claim that MLX is faster than PyTorch MPS, that 4-bit models always fit a given Mac, or that the completion is correct. The response stopped at the declared 64-token limit and was not quality-scored.

The fixed prompt exercises chat-template rendering, tokenization, generation, and stop behavior without claiming to measure reasoning, factuality, safety, or instruction following. It ends at the configured length limit, so completion length is a workload parameter rather than a quality signal. A future quality evaluation needs a versioned prompt set, scoring rubric, and human or task-based judgments.

This distinction changes product choices. A private, offline note-search tool may value predictable local availability even if its first response is slower than a hosted alternative. A customer-facing assistant may need a model quality evaluation, request queueing, observability, and a fallback policy before its runtime is selected. The benchmark does not choose for us; it supplies one traceable input to that choice. Keep user data, prompts, logs, and generated text within the same privacy review: running weights locally does not by itself guarantee that every surrounding service or log is local.

Worked workload: what the recorded run does—and does not—tell us

The recorded run begins after the selected public model is already available in the local cache. The script loads mlx-community/Qwen2.5-0.5B-Instruct-4bit, applies its one-message chat template to the fixed seed-record prompt, warms up with four unmeasured tokens, resets the MLX peak counter, then streams at most 64 new tokens at temperature zero. The rendered prompt contains 43 tokens. This is a precise workload, not a generic statement about "running a half-billion-parameter model on a Mac."

The trace separates a 542.927 ms model-load observation from the 290.701 ms measured generation. The final response reaches the configured 64-token length limit and reports 342.260 generated tokens/s. The immediate engineering interpretation is limited: this package/model/prompt combination completed that warmed-up, length-capped stream under the recorded conditions. It says nothing about first-token latency, a cold download, a 4,000-token context, concurrent requests, answer quality, or another quantization.

Now inspect memory without adding incompatible fields. Before load, the Python process RSS was 333,725,696 bytes; after load it was 785,121,280 bytes; before measured generation it was 796,934,144 bytes. After warm-up, MLX reported 281,784,072 active bytes and 345,934,496 peak bytes, while MLX-LM reported 0.345934496 GiB peak memory. These are labeled allocator and process views, not pieces of an arithmetic total. The correct conclusion is that the record contains several reproducible observations; it does not establish the maximum model or context that will fit a particular Mac.

To turn this smoke test into a capacity experiment, keep the model, cache state, temperature, output cap, and background workload declared. Run a small series at one prompt length, then increase only prompt length while retaining each raw record. Stop when an explicit threshold is reached—for example an error, an observed pressure event, or an application-defined latency limit—and report the first failing configuration as well as the successful one before it. Changing model size, quantization, prompt length, and concurrent apps at once would produce an anecdote rather than a diagnosable result.

What broke

The first issue was dependency compatibility: MLX-LM 0.31 requires Transformers 5, while the first Chapter 6 draft capped Transformers below 5. The project resolved this by upgrading the optional Transformers group and rerunning the Chapter 6 MPS and CPU checks before relying on it. This is why optional groups still need an integrated lockfile, not isolated installation instructions.

Model download is another boundary. The first run uses the Hugging Face cache outside Git and may show an unauthenticated-request warning when no Hugging Face token is configured. A token can improve download limits, but no token or credential is needed for this public model and none belongs in the repository.

Finally, a memory field is not a capacity guarantee. Unified-memory pressure can depend on other work running on the computer, caches, and longer prompts. Check available storage before downloading models and use an explicit workload before increasing model size or context length.

Classify a local failure before changing frameworks. A missing optional package is an installation problem; a cache/download error is a storage or network problem; a chat-template error is a model-interface problem; memory pressure is a workload/capacity problem; and an unhelpful completion is an evaluation problem. Each needs a different record and remedy. Collapsing them into "the model does not work on my Mac" makes the project impossible to reproduce.

When memory pressure appears, preserve the failing configuration before trying a smaller model. Record the prompt and output limits, the model identifier, whether other demanding applications were running, the final error or observed symptom, and the three labeled memory values when available. Then reduce one variable, rerun, and keep both records. This avoids a misleading story in which an apparently magical configuration change fixed a failure whose real cause was an unrelated browser tab, a cold cache, or a longer prompt.

Alternatives and when to use them

Use MLX-LM when you want a native local inference path and can make model, quantization, cache, memory, and evaluation limits explicit. Use a hosted API when a model, scale, or operational capability is unavailable

locally—but keep credentials environment-configured and preserve the same evidence discipline. PyTorch MPS is a valuable alternative for models and training workflows already expressed in PyTorch. Do not choose on a slogan: choose after measuring the actual workload.

Smaller task-specific models can be better local choices than a general instruction model for classification, extraction, or embedding. Quantized GGUF-style toolchains are another alternative with their own formats and measurement rules. A remote service can simplify distribution but changes privacy, cost, latency, and outage behavior. State which trade-off matters before choosing a runtime.

A practical selection checklist is short: identify the task and quality bar; declare whether data must remain local; measure cold and warmed behavior at the expected prompt and output sizes; observe storage and memory headroom; and define what the application does when the model is unavailable or its answer is not supported by evidence. The Book Intelligence Assistant later applies the last rule directly: retrieval can fail honestly, and a planner must stop before writing. Local inference is a component of that policy, not a replacement for it.

Evidence trail

The local-inference note is research/04-mlx/notes.md. Run experiments/07-mlx/run_local_model.py only after the documented optional install, then interpret its one workload through benchmarks/04-mlx/README.md.

Takeaway

Local inference is a systems decision, not only a privacy preference. The responsible first question is not “how many tokens per second?” but “which model and quantization, on what prompt, with what cache, memory method, warm-up, and quality boundary?”

Run the lab on your Mac

QR code for Chapter 07 lab

Online lab: https://hassanvfx.github.io/mac-ai/labs/07

Repository: experiments/07-mlx/run_local_model.py

```
uv run --group mlx python experiments/07-mlx/run_local_model.py
```

Expected: Local-model observation with recorded settings.

Evidence: benchmarks/04-mlx/README.md

Scan the code to open the live lab, then use Safari Share → AirDrop to send it to your Mac. The lab follows main; clone the repository once and run the command from its root directory.

Turning Meaning Into Geometry

Intuition

An embedding is a numerical coordinate for an item such as a sentence. If an encoder has learned useful regularities, nearby coordinates can represent texts that are related even when they do not share the same words. That is not magic: it is a retrieval heuristic whose output still needs inspection. Sentence-BERT made this practical by training a BERT-derived model to produce sentence-level vectors suited to similarity comparison (Reimers and Gurevych 2019).

Similarity changes the first question from "which exact words occur?" to "which stored passages might help?" It does not change the second question: whether a returned passage supports the answer. The assistant therefore treats a vector score as a routing signal and retains the evidence a reader needs to inspect.

In geometric terms, an encoder maps text to a point in a high-dimensional space. L2 normalization makes each nonempty vector have length one, so the dot product measures the angle between two directions: identical directions score one, orthogonal directions score zero, and opposite directions score below zero. The implementation uses that relationship because it is easy to inspect, not because a cosine score has an intrinsic meaning such as "80% correct." A score only ranks candidates produced by this encoder over this corpus.

Problem

Before adding a vector database, framework, or language model, we need to answer a smaller question: can this book find its own evidence? The corpus is versioned Markdown, code, and benchmark records. A useful result must retain the source path, corpus kind, chapter identifier, citation keys, and relevant experiment or benchmark grouping—not merely a float score.

Chunking is the first practical design decision. A whole chapter is usually too broad: it can match a query while leaving a reader with thousands of words to inspect. A sentence can be too narrow: it loses the condition or citation that makes the statement meaningful. The baseline therefore joins paragraphs up to a bounded character budget. The boundary is deliberately

plain and versioned; it can be tested, changed, and recorded in a later benchmark rather than being hidden inside a hosted retrieval service.

Metadata belongs to the chunk that was ranked, not to a guessed answer. If the chunk came from a benchmark, its record group stays attached. If it came from a numbered chapter, the chapter identifier stays attached. If it contains a citation key, that key travels with the result as a clue for the reader to check in the shared bibliography.

Chunk boundaries are an information-retrieval policy, not cleanup trivia. A boundary that separates a benchmark number from its workload can produce an apparently relevant but unusable result. One that merges several unrelated sections can hide which statement the score actually matched. The baseline uses blank paragraphs because Markdown authors already use them to group an idea, and its limit=1200 character rule is visible in src/from_tensors_to_agents/book_intelligence.py. This is a practical default, not an ideal universal size. Change it only alongside a fixture evaluation and record the corpus revision that produced the result.

There are two useful kinds of determinism here. The hash-vector baseline is deterministic across runs of the same code and text, so a failing attribution test has a narrow debugging surface. A learned encoder can also be made repeatable enough for an experiment by pinning its model identifier, package versions, corpus snapshot, and ranking procedure. Neither form proves that the ranking is good. Repeatability tells us that we can investigate a result; it does not turn a similarity score into relevance.

Minimal implementation

The baseline in book_intelligence.py splits allowed repository files into bounded paragraph chunks. It hashes tokens into a fixed-size vector, normalizes it, and ranks query/chunk dot products. This deterministic method is intentionally crude, but it is small enough for fixtures and proves the data contract without a model download.X

The hash is not a learned representation of language. It maps token counts into a fixed vector using a stable digest and then normalizes the result. Collisions, synonyms, and word order make it unsuitable for claims about semantic quality. Its value is different: the same fixture and query produce the same ordering, so tests can verify chunking, path attribution, citation propagation, empty input, and refusal behavior on a CPU-only machine.

The fixture is deliberately smaller than the live book corpus. It is a frozen, non-sensitive set of source-shaped files whose expected paths and citation keys belong in tests. That gives automated checks a stable oracle even while this manuscript changes daily. The live corpus is the useful authoring workspace; its generated index remains local because it reflects the current checkout and may be stale the moment another chapter is edited. Never treat a passing fixture ranking as proof that the live corpus has complete coverage.

```
uv run python experiments/08-embeddings/book_search.py --deterministic \
  --query 'Where are benchmark limitations recorded?'
```

The index written under .book-intelligence/ is a generated local artifact. It is useful for inspection and deliberately ignored by Git.

Real implementation

The optional learned path uses sentence-transformers/all-MiniLM-L6-v2 through learned_retrieval.py. It encodes each existing Evidence chunk locally, L2-normalizes the rows, and uses a dot product as cosine similarity. Crucially, it does not replace the evidence record; every ranked result carries the original provenance through the learned index.X

The learned and deterministic indexes share the Evidence contract. Both return a source path, kind, chapter, citations, text, and metadata alongside a score. This lets later RAG and planning layers work against an explicit interface rather than assume a particular vector database. If the learned encoder cannot be installed or loaded, the fallback must label itself instead of implying that learned search took place.

The implementation normalizes the complete matrix of document vectors and the query vector before their dot product. This gives cosine similarity for nonempty vectors, but the score remains relative to this corpus and encoder. Do not compare a score from one embedding model or chunk policy with a score from another as though they were calibrated probabilities. Compare outcomes: did an acceptable evidence path appear in the returned set, did the result retain its source metadata, and could a reader inspect the supporting passage?

The encoder is optional for two operational reasons. Downloaded weights and their cache consume local storage, and the model may be unavailable on an offline or CPU-only machine. The command begins with df -h . so an

author can check storage before triggering that download. If the optional path fails, use the deterministic baseline to keep contract tests and the rest of the book workflow working; report the learned run as unavailable rather than silently substituting a different model.

```
df -h .
uv sync --group embeddings
uv run --group embeddings python experiments/08-embeddings/book_search.py \
  --query 'Where do we record benchmark timing limitations?'
```

The first learned run may download public weights into a local Hugging Face cache. The cache is not the corpus and must never be committed. The recorded project-machine observation, including package/model identity and an explicit statement of what was not measured, is in benchmarks/05-book-intelligence/README.md in the companion repository.

Experiment

Ask both backends the same question and compare the *paths*, not just the top score. For this repository, a useful neighbor should lead a reader to a benchmark, research note, chapter, or runnable script that can be checked. Record the exact query, corpus revision, model identifier, returned paths, and any misleading neighbor. The first learned run is a correctness observation; it is not a retrieval-quality, latency, memory, or accelerator benchmark.

Build a small evaluation set before tuning retrieval. For each question, record one or more acceptable source paths, the reason each is relevant, and any path that would be dangerously misleading. Run the same set after changing a chunk limit, embedding model, normalization rule, or corpus snapshot. Separate retrieval recall from answer grounding: retrieval asks whether useful evidence appeared; grounding asks whether the response cites only evidence it received.

Use simple measurements before elaborate dashboards. For a question with an acceptable path set, recall at k asks whether at least one acceptable path is in the first k returned results. A path-attribution check asks whether every displayed result actually resolves inside the corpus. A failure review asks why the miss occurred: query vocabulary, an over-broad chunk, a missing document, or a misleading but lexically similar neighbor. These measures expose different problems and should be stored with the question rather than reconstructed from a favorable terminal screenshot.

The retrieval interface is intentionally read-only. It can assemble evidence for a later plan, but it cannot edit a chapter, rewrite a benchmark, or alter Git state. This boundary keeps the first capability easy to audit: input files produce chunks; chunks produce ranked candidates; candidates preserve paths. Chapters 9 through 12 add answers, structured plans, and approval state on top of this contract without granting retrieval itself write authority.

Worked retrieval audit: from a question to inspectable evidence

Take the fixture question, "Where is cosine similarity explained?" The right way to judge the result is not to celebrate a number in the terminal. First, the evaluator builds an index only from its frozen research/, book/chapters/, experiments/, and benchmarks/ directories. It deliberately does not crawl the parent checkout, a home directory, or an arbitrary path named in a Markdown link. That bounded input is what makes a returned relative path meaningful.

Second, it asks for a small ranked set and checks that evals/fixture_corpus/research/embeddings.md occurs among the candidates. That fixture source says what normalization and cosine comparison mean and carries the reimers2019sentencebert citation key. A useful display therefore includes the path, excerpt, and key—not just "similarity score: 0.42." The reader can open the note, locate the cited claim in research/references.bib, and decide whether it answers the question. If the model had instead returned only a benchmark record mentioning the word *similarity*, that would be a candidate to inspect, not evidence that the explanation was found.

Third, repeat the same query against the unchanged fixture. The deterministic baseline must return the same ordered paths. This is a regression property, not a semantic-quality result: hash collisions can still make the order unhelpful. Its practical value is that a changed order now has a short list of possible causes—fixture text, tokenization, chunking, vector dimensions, or the tie-break rule—rather than an unexplained hosted-service change.

Finally, probe the refusal edges. A blank query produces an all-zero query vector, so the retriever returns no candidates and the grounded-answer layer must say that no indexed evidence matched. An absent corpus produces an

empty index and the same refusal. A deliberately unsafe fixture link to a path above the corpus root is reported by the reviewer as an escaped link; it is never followed. These cases make the boundary concrete: retrieval can name evidence inside its declared corpus, but it cannot invent an answer or use a link as permission to read elsewhere.

Run the frozen audit with:

```
uv run pytest tests/test_book_intelligence.py \
  tests/test_book_intelligence_evaluation.py tests/test_reliability.py
uv run python evals/run_reliability.py
```

The expected result is that the versioned cases pass. It is not an expected retrieval score, a claim that learned embeddings are installed, or a claim about the quality of a model on a different corpus. When replacing the hash baseline with the optional encoder, retain this exact audit shape and add a separate versioned relevance set before asserting that learned retrieval is an improvement.

What broke

Three failure modes matter immediately. Empty input has no evidence and must not turn into a confident answer. A mathematically high score can still be a bad neighbor because the chunk is broad or the query is vague. Finally, a citation key found in a chunk says that the chunk contains a citation; it does not automatically support every sentence in the chunk. The tests cover empty indexes, deterministic rankings, metadata preservation, and weak-result refusal. The research note records the remaining limits (Reimers and Gurevych 2019).

Stale indexes are another failure mode. A local index is a cache of tracked files at a particular moment, not an authority over the current working tree. Rebuild it after editorial changes, and resolve every returned path inside the configured corpus before displaying it. The corpus reviewer rejects Markdown links that escape that boundary.

Similarity can also encode an accidental shortcut. A query asking about “memory” may retrieve a local-inference benchmark when the author meant a LangGraph checkpoint, because both use the same word. A query with a citation key may be better served by exact lookup than embedding search. Preserve the query and the returned excerpt in an evaluation failure fixture;

saying only that “retrieval was bad” leaves no way to tell whether an embedding change, chunk boundary, or task wording should change next.

Alternatives

Keyword search is cheap, predictable, and sometimes best for exact filenames or citation keys. Sparse BM25-style retrieval, hybrid lexical-plus-vector retrieval, rerankers, and a dedicated vector store are reasonable later alternatives. They add operational cost and do not remove the need for source attribution. For a small changing book corpus, a transparent in-process index is the right first control.

Use exact search first when the reader knows a section name, experiment path, or BibTeX key. Use a hybrid approach when exact identifiers and paraphrased questions are both common. Consider a persistent vector store only when corpus size, update rate, filtering needs, or multi-user access makes an in-process index impractical. In every case, keep the index revision, chunking policy, and returned paths inspectable before optimizing ranking sophistication.

Reranking is useful when a cheap first pass returns many candidates but the top few need finer discrimination. It is not a free accuracy button: it adds a second model or rule, another versioned dependency, and another place to lose provenance. Add it only when a recorded evaluation miss justifies it, and make the reranker return the original Evidence records rather than a detached summary. For this book-sized corpus, transparent candidates are more valuable than an opaque score improvement.

When to use it—and when not to

Use semantic retrieval to discover candidate evidence when wording differs between the question and the source. Do not use similarity as proof, a substitute for reading the returned source, or a performance claim about a model. When an answer needs exact legal, safety, or publication requirements, retrieve the primary source and inspect it directly.

Evidence trail

Read research/05-embeddings-and-rag/notes.md, run experiments/08-embeddings/book_search.py, and inspect the local model observation and limits in benchmarks/05-book-intelligence/README.md.

Takeaway

Embeddings make meaning searchable, not self-verifying. The important product of this chapter is a provenance-preserving retrieval contract that later RAG and agent workflows cannot quietly discard.

Run the lab on your Mac

QR code for Chapter 08 lab

Online lab: https://hassanvfx.github.io/mac-ai/labs/08

Repository: experiments/08-embeddings/book_search.py

```
uv run python experiments/08-embeddings/book_search.py --deterministic
```

Expected: Evidence-preserving search results.

Evidence: benchmarks/05-book-intelligence/README.md

Scan the code to open the live lab, then use Safari Share → AirDrop to send it to your Mac. The lab follows main; clone the repository once and run the command from its root directory.

RAG: Giving Models a Memory They Never Learned

Intuition

Retrieval-augmented generation (RAG) changes what a model can see at answer time. It does not retrain the model or make it truthful. A good RAG system therefore begins with a more modest promise: retrieve evidence, show where it came from, and say when there is no evidence.

That promise separates three stages that are often collapsed: retrieval finds candidate chunks, grounding constrains output to those chunks, and generation turns grounded material into prose. A fluent response can fail at any stage. This chapter keeps generation out of the control path so a reader can inspect the evidence before trusting a summary.

The phrase "model memory" is convenient but misleading. The model weights do not absorb the book when an index is built, and a retrieved passage is not a new fact in the model. Retrieval selects text for the current context. When the book changes, rebuild the index, retrieve again, and inspect the current source rather than assuming an earlier answer remains correct.

Problem

The Book Intelligence Assistant must answer questions about this changing repository without inventing facts about experiments, benchmarks, or chapters. Its first RAG stage should be inspectable without an API key or a generative model. That gives us a control before we add prompt templates and agent graphs.

The central design challenge is to keep the boundary visible: distinguish the evidence packet, a claim directly supported by that packet, and a helpful sentence that goes beyond it. If these objects are blended into one fluent response, a reader cannot tell whether a citation supports the statement it appears to support or merely sits somewhere nearby in the context window.

Think of the packet as a small case file. Its job is not to persuade; its job is to preserve a chain from a question to an inspectable repository artifact. The

question chooses candidates, each candidate retains its path and local metadata, and the reader can compare the excerpt against the eventual claim. That makes disagreement productive. A reviewer can say “this chunk does not establish that conclusion” and point to a concrete boundary, rather than argue with an opaque answer generated from an unknown context window.

Retrieved text is also data, not trusted instructions. A Markdown note can contain an outdated command, a quoted prompt, or a sentence asking a future assistant to behave differently. None of it may override the assistant’s read-only and approval policies. Keep the corpus directory allowlisted, treat its contents as evidence to cite, and make state-changing behavior impossible until a separate approval checkpoint has been reached.

Minimal implementation

grounded_answer.py builds the same corpus index from Chapter 8 and formats only the retrieved excerpts. It places the repository path and any citation keys next to each excerpt. If no deterministic result has positive similarity, it returns an explicit missing-evidence message.X

```
uv run python experiments/09-rag/grounded_answer.py --deterministic \
  --query 'What should an experiment record?'
```

This is intentionally not a conversational answer. It is a reader-visible evidence packet: the safe object that a later model may summarize only under a grounding policy.

The packet format is intentionally simple: a heading saying it is grounded evidence only, followed by each repository path, any citation keys found in that same chunk, and the unaltered excerpt. It is not elegant prose, but it makes auditing easy. A reader can open the named file, compare the excerpt, and decide whether the question was actually answered.

The format makes a useful negative result possible. “No grounded answer” means the configured index did not return evidence that meets the declared rule. It does not mean the repository, the Internet, or the world lacks the answer. A reader can then reformulate the query, use exact search, add a missing source to the corpus, or investigate outside the corpus through an appropriate primary source. Refusal preserves this next step; an invented summary would conceal which evidence is actually missing.

Real implementation

With the optional local encoder installed, the same program uses the learned index from learned_retrieval.py. It applies a conservative score threshold before formatting evidence. A weak or empty result fails closed rather than producing a plausible paragraph.X

```
uv run --group embeddings python experiments/09-rag/grounded_answer.py \
  --query 'What should an experiment record?'
```

The current output remains excerpt-only. That boundary is deliberate: it lets us test retrieval, citation propagation, and source paths independently from a language model's writing quality. The current implementation and limitations are recorded in benchmarks/05-book-intelligence/README.md and research/05-embeddings-and-rag/notes.md in the companion repository.

The learned path applies its threshold after ranking. This is a rejection rule, not a truth detector: it says that this backend did not find a candidate strong enough for the current conservative policy. Thresholds need evaluation data; raising one can hide useful evidence, while lowering one can turn weak neighbors into authoritative-looking context. The correct response to either uncertainty is to display the evidence or refuse, not to smooth it into confidence with a language model.

Context also has a budget. Passing every retrieved chunk to a generator can dilute relevant material, exceed a context limit, or let a tangential excerpt appear to support an unrelated sentence. Start with a small declared limit and keep ranked paths visible. If later compression is needed, retain a link to the original chunk and test that a qualification or limitation was not removed.

If a generator is later added, give it a strict output contract: cite each claim with the retrieved repository path that supports it; mark synthesis or uncertainty; and return the missing-evidence response when the packet cannot support an answer. The generator should never fabricate a path to make prose look grounded. A post-generation verifier can check that every cited path came from the packet and resolves inside the current checkout, but that structural check is not a substitute for a reviewer reading whether the cited passage actually entails the claim.

Context assembly is a separate experiment from retrieval. Record which k chunks were selected, their order, any truncation, and the final prompt or structured input given to a model. If an answer loses a benchmark caveat,

this record lets us determine whether the caveat was never retrieved, was discarded during packing, or was ignored during writing. Without it, a RAG failure gets misattributed to "hallucination" even when the relevant source never reached the model.

Experiment

Use a question whose answer has a known source path. Confirm that each rendered path exists in the checkout, inspect each excerpt, and mark whether it actually answers the question. Then ask an unsupported question and verify the refusal. The versioned evaluation set will grow from these cases: locating evidence, spotting an unsupported statement, proposing an experiment, and reviewing a chapter change. A retrieved path is auditable evidence; an uncited answer is not.

Test failure cases as carefully as successful lookups. An empty corpus or blank query should yield no packet. A weak learned result should yield the same explicit refusal, not a low-confidence paragraph. A known citation key should survive indexing and appear next to the retrieved chunk. A rendered relative path must resolve inside the configured checkout. These are structural properties that remain testable while the project has no selected API provider.

For a future generative layer, score two things separately. First, did retrieval include evidence that answers the question? Second, did the generated answer make only claims that its cited excerpts support? Keep source revisions and human judgments in a versioned dataset rather than choosing only favorable demonstrations.

Add a third evaluation category for citation precision. A response may name a real file and still cite it beside the wrong assertion. For each test case, store the expected path set, a short statement of what it supports, and one plausible-but-insufficient neighbor when appropriate. Reviewers can then score whether the response used the right evidence, not merely whether it displayed some evidence. The current no-model cases exercise the structural half of this policy: source-path attribution, citation-key preservation, missing-evidence behavior, planning no-write behavior, and review findings.

Quality evaluation needs honest boundaries too. Do not convert one attractive answer into an accuracy rate. Fix a small question set before prompting, preserve failures and refusals, identify the corpus revision, and distinguish human judgment of factual support from style preference. If a

selected API or local model is unavailable, log that condition and run the deterministic evidence tests; do not silently replace the evaluation with an incomparable provider.

Worked grounded-answer audit: a packet is not a conclusion

Use the frozen corpus question, "Where is cosine similarity explained?" The deterministic retrieval layer returns a short ranked set. The grounded-answer function then performs a deliberately boring transformation: it drops non-positive candidates, prints a Grounded evidence only heading, and emits each remaining excerpt beneath its repository path. When a chunk contains a citation key, that key appears beside the same path. It does not infer a one-sentence answer such as "cosine similarity is the best retrieval metric," because the packet does not establish that claim.

An auditor follows four checks. First, each displayed relative path must be inside the configured corpus and resolve to a real current file. Second, the excerpt must actually contain the material being offered as evidence; a path alone is not enough. Third, a citation key must have travelled from that exact chunk rather than being copied from another result. Fourth, the auditor decides whether the excerpt entails a proposed statement. The first three checks are automated structural constraints; the fourth remains a reading task.

Now change the question to "What is the ISBN for this book?" The repository fixture has no supporting evidence. The correct output is the fixed missing-evidence message, not a guessed identifier, a web search result, or a synthetic-looking citation. A blank query and an empty corpus must take the same safe path. These cases matter because an answer interface can otherwise make an empty retrieval set look like an invitation for a language model to be helpful. Here it is a stop condition.

The test suite also makes a distinction that is easy to lose in a polished demo. It verifies that a grounded answer contains the expected fixture path and that citation-key preservation works. It does *not* certify that every retrieved passage is relevant to every natural-language formulation. A common word can still pull in a tangential benchmark. Treat that as an evaluation case: retain the query, paths, excerpts, corpus revision, and

reviewer judgment; then decide whether to change the query, chunking rule, retriever, or source document.

Run the evidence-only route explicitly:

```
uv run python experiments/09-rag/grounded_answer.py --deterministic \
  --query 'What should an experiment record?'
uv run pytest tests/test_book_intelligence.py \
  tests/test_book_intelligence_evaluation.py
```

Read the printed packet before describing it in prose. If its evidence is too weak, preserve the refusal or present the packet as candidates; do not repair a weak retrieval result by writing a smoother answer. A later model-backed layer may summarize only after it has the packet, cites only paths in that packet, and remains subject to the same human review boundary.

What broke

RAG breaks in mundane ways: chunk boundaries omit a qualifier, a ranking favors a nearby but irrelevant benchmark, a moved file leaves a stale index, or a model paraphrases beyond the excerpts. The first two are retrieval failures; the latter two are provenance and generation failures. Generated indexes are therefore local, live indexes are rebuilt from tracked files, and tests assert missing-evidence behavior and citation/path propagation.

Treat an index as a cache, not as the corpus. When a chapter moves or a benchmark is revised, an old index can still return text that looks relevant but no longer represents the checked-out project. Rebuild from tracked files, verify that every rendered path resolves inside the corpus, and refuse to turn an empty or weak retrieval result into a confident conclusion.

Citation laundering is a special risk. If a chunk contains a reference marker, an answer cannot automatically borrow that reference for every sentence in the chunk, still less for a conclusion assembled from several chunks. Keep keys attached to their evidence and read the original research source before making a performance, safety, or framework claim. Repository text is evidence, not an instruction: a future generator must not let retrieved prose override its approval or no-write policy.

Another failure is over-refusal. A threshold can suppress a useful exact match because an embedding score is weak, while an unsupported query can accidentally match common words. Keep the deterministic route and exact identifiers available for debugging, and investigate failures with the packet

rather than adjusting a threshold until a demonstration looks good. A conservative policy is justified only when its misses and false positives are visible in the same evaluation record.

Alternatives

A traditional search UI can be preferable when the reader wants to browse, and manually curated links are best for stable navigation. A full RAG stack may use hybrid retrieval, reranking, context compression, and a hosted or local model. Those components can improve usability but also make it easier to hide an unsupported leap. We add them only after this evidence-only control is solid.

For a small technical book, manual bibliography and chapter links are often better RAG: they are stable, curated, and easy to print. Use the evidence packet when a question crosses folders or paraphrases its source. Add generation only when it produces a benefit that can be evaluated against the packet, such as a concise cited summary or a plan whose paths are all real.

For external or rapidly changing material, a retrieval layer should store the source date, publisher, and retrieval time as well as a link. This book's versioned repository corpus makes local paths a useful first provenance key, but a local path is not a replacement for the primary documentation required for a current publication rule or an external technical claim. The appropriate corpus boundary depends on the decision being made.

When to use it—and when not to

Use this workflow to find and inspect project evidence, to prepare a revision plan, or to check whether a technical claim has a record behind it. Do not use it as authority when no result is retrieved, when the cited source is stale, or when a decision needs a primary external source. In later chapters, any proposed modification still stops for explicit human approval.

Do not silently turn an evidence packet into a final answer in a high-stakes setting. For external publication requirements, licensing, medical, legal, or security questions, retrieve the primary source, state its scope and date, and ask the responsible person to review it. RAG is context management, not an authority transfer.

Evidence trail

The RAG source note is research/05-embeddings-and-rag/notes.md; run experiments/09-rag/grounded_answer.py and the versioned cases in evals/book_intelligence.jsonl before treating a retrieved excerpt as support.

Takeaway

RAG is not a memory implant. It is a disciplined context pipeline. The first safe version does less—returning evidence rather than fluent prose—so that the rest of the system has something trustworthy to build on.

Run the lab on your Mac

QR code for Chapter 09 lab

Online lab: https://hassanvfx.github.io/mac-ai/labs/09

Repository: experiments/09-rag/grounded_answer.py

```
uv run python experiments/09-rag/grounded_answer.py --deterministic
```

Expected: Grounded answer with repository evidence.

Evidence: benchmarks/05-book-intelligence/README.md

Scan the code to open the live lab, then use Safari Share → AirDrop to send it to your Mac. The lab follows main; clone the repository once and run the command from its root directory.

From LLM Calls to AI Systems

Intuition

An LLM call becomes a system when its input, output, failure behavior, and permissions are explicit. An orchestration library can make integration easier, but it cannot turn ungrounded context into evidence or a parsed JSON object into permission to act.

The useful unit of comparison is an adapter contract. Given the same retrieved evidence and requested schema, each adapter should either return a validated, source-scoped object or expose why it cannot. Provider choice, prompt wording, and parsing convenience are secondary to that invariant when the result may influence a chapter or experiment plan.

Structure is a data contract, not a truth contract. A schema can confirm that an output has fields called evidence_paths and steps; it cannot establish that a cited path exists, that an excerpt supports a claim, or that a suggested step is appropriate. The adapter must validate model-shaped data against the evidence the retrieval layer actually supplied. Untrusted input crosses a boundary, is checked, and is rejected or normalized before downstream code relies on it.

Problem

The Book Intelligence Assistant needs to turn retrieved source material into a chapter-improvement plan and a critique. We want to compare a direct API path and LangChain without changing the evidence, hiding parsing errors, or giving either path write access.

Both adapters receive the same objective, allowed source paths, excerpts, schema, and no-write instruction. If one route retrieves different context or accepts different paths, a difference in its final plan cannot be attributed to the SDK/framework choice. Keep provider/model comparisons separate until a provider is selected and a controlled benchmark records its conditions.

This is an experimental control rather than a contest between two brand names. Fix the task, retrieved evidence, schema version, decoding settings, retry policy, and clock boundary before comparing a configured provider run. If the answers differ, inspect the context and validation traces before

calling either route more reliable. A framework can introduce defaults, while a direct client can expose details a framework normally manages; neither fact predicts answer quality on its own.

Separate configuration from content. The objective and evidence arrive from the caller. Endpoint, model identifier, and credentials arrive from the environment only when an optional API experiment is explicitly requested. Package versions, model names, retry behavior, redaction decisions, and timing method belong in the resulting benchmark record. That separation prevents a chapter draft or test fixture from becoming an accidental vehicle for a secret or a provider-specific production decision.

Minimal implementation

structured_planning.py defines Pydantic schemas for a proposal and review. Both receive only retrieved paths and excerpts. Validation drops paths that were not retrieved, records a warning, clears action-like steps when no evidence exists, and forces approval_required to true.X

The direct route is deliberately thin: an OpenAI-compatible client reads three environment variables only when requested, sends the schema, and returns the parsed object. Missing configuration fails before a network call.

The plan schema includes an objective, evidence paths, steps, unsupported-claim warnings, and approval_required. The review schema includes the same objective and paths plus findings. After parsing, validation discards every path outside the retrieved allow-list, appends a warning, and replaces the model's objective with the caller's objective. With no retrieved evidence, it clears plan steps and adds a warning instead of accepting a well-formed but unsupported proposal. It forces approval to true even if a model emits false.

This normalization makes authority explicit. A model can propose text; it cannot widen the files it may cite, redefine the request it received, or waive human review through a field in its own output.

The validation order matters. First parse the shape; then compare every path with the retrieval allow-list; then overwrite caller-controlled fields such as the objective and approval flag; finally decide whether the object has enough evidence to be useful. A response that is valid JSON but includes outside.md is an authorization failure, not a successful plan with one small warning. This project retains the permitted portion for inspection and

records the rejected path, but it never treats the discarded citation as support.

No evidence has a stronger consequence. The plan schema may still parse, but the implementation clears its steps and appends a missing-evidence warning. That prevents a fluent model from converting an empty retrieval result into an apparently actionable edit list. A human may use the objective to search again or add research, but the system has not earned a grounded recommendation.

Real implementation

LangChain's ChatOpenAI integration exposes a similarly structured runnable through with_structured_output. The implementation uses include_raw=True so parsing failures remain visible instead of being silently converted into a plausible result. The two routes are compared over the same evidence list, not over separately retrieved contexts. LangChain documents structured output for this integration (LangChain 2026a).

Install the optional group and run the no-network fixture comparison:

```
df -h .
uv sync --group agents
uv run --group agents python experiments/10-systems/compare_structured_planning.py
```

The --api flag is deliberately separate. It requires BOOK_INTELLIGENCE_API_KEY, BOOK_INTELLIGENCE_API_BASE, and BOOK_INTELLIGENCE_MODEL; credentials belong in the process environment, never in a source file, shell history, benchmark record, or commit.

The environment boundary is intentional. The project does not choose a public endpoint or a model for the reader, and it does not need an API key for the deterministic test suite. A configured adapter must still declare endpoint type, model identifier, retry policy, redaction policy, and timing boundary in a benchmark record before its output can support a comparison claim. Loading a key is not authorization to make an uncontrolled experiment or a repository change.

Use a narrow environment file or shell session for a controlled run and keep it outside version control. Do not print the key while diagnosing a failure; log only the non-secret configuration fields needed to reproduce the experiment. If the base URL, model, or credential is absent, the adapter raises

a configuration error before a network request. That explicit failure is better than a silent fallback to an unknown default model, whose output could later be mistaken for a recorded comparison.

The optional local path follows the same discipline. A local endpoint or model is still a configured provider with a model revision, context policy, and failure modes. “Local” describes a deployment location, not an exemption from schemas, provenance, evaluation, or approval. The deterministic fixture run is the baseline that remains available when neither a local model nor an API is configured.

Experiment

The recorded fixture run proves a narrow invariant: both adapters receive the same retrieval result and return the same allowed source path. The regression tests then send an invented path, an empty evidence set, malformed structured output, and absent configuration. These are system-contract tests, not a comparison of model intelligence or provider quality. See benchmarks/06-structured-systems/README.md for the package versions and limits.

Read the comparison output as a contract trace. It prints the paths preserved by the direct and composed adapters and whether both demand approval. It does not say their natural-language plans are equivalent, because the default run uses fixtures rather than a remote model. The tests exercise an invented outside.md path, an empty evidence list, malformed structured output, and absent API variables. A future model experiment must hold task set, model, settings, and retrieved context fixed before interpreting a difference between adapters as overhead or reliability.

Treat malformed output as a first-class test case. It can be invalid JSON, missing fields, a response object with a parsing error, an unexpected scalar, or a formally valid schema that contains an unapproved evidence path. The direct and LangChain adapters must expose the failure rather than retry until they get a convenient result. Retrying may be appropriate in a measured production policy, but it changes cost, latency, and the probability of a different answer, so it must be declared and evaluated.

The fixture comparison does not measure network latency or token use. It proves that a composed adapter does not weaken the evidence allow-list or human approval requirement. Once a provider is chosen, record separate timing observations for request construction, remote response, parsing, and

validation where those boundaries matter. Do not use a no-network unit test to support a claim about an API's speed, reliability, or model quality.

Worked validation: a well-formed but unsafe plan

The fixture responder receives one allowed result: book/chapters/08-turning-meaning-into-geometry.md. It returns an object that looks valid: it has an objective, an evidence-path list, one implementation step, no warnings, and approval_required: false. But the path list also adds outside.md, and the model-supplied objective does not match the caller's request. A parser that only checks JSON shape would accept this object and make it look authoritative.

validate_plan applies the actual contract. It keeps only the retrieved Chapter 8 path, records Rejected unsupported evidence paths: outside.md, replaces the objective with the caller's objective, and forces approval_required to true. The surviving step remains a proposal, not a write. This is a useful distinction: the response was syntactically valid but failed an evidence/authority check. A schema did its job only after the application compared its fields with trusted context.

Run the same responder with no retrieved evidence. The path allow-list is empty, so the validator clears every plan step and appends the warning "No retrieved evidence: do not make an implementation claim." It does not invent a generic safe-sounding task list. The caller can retrieve again, narrow the objective, or stop; it cannot present the model-shaped object as an evidence-backed recommendation.

The LangChain fixture follows the same exercise through a different adapter. It returns a parsed structured object through with_structured_output, then passes it to the same validator. A parsing error or missing parsed object raises instead of becoming a guessed fallback. The two routes therefore share the important behavior even though their client code differs: only the supplied evidence may survive, approval cannot be waived by the response, and malformed output remains visible to the operator.

What broke

A valid schema can still contain an unsupported citation, a plan can sound reasonable with no evidence, and a compatible-looking endpoint can lack the

structured-output behavior assumed by an adapter. The safe response is to filter against the retrieved paths, preserve the parse error, and stop. Do not recover by inventing a fallback answer or silently changing the requested schema mode.

Parser recovery is not correctness. A framework may retry, coerce a field, or return a partially parsed value; those behaviors can be useful, but they must be visible and followed by the same evidence validation. This project chooses failure over a guessed plan whenever structured output is absent or malformed. A reader can then fix configuration, adjust a declared schema, or use the deterministic path without pretending the model supplied grounded output.

Schema evolution is another quiet failure mode. Adding a required field, renaming a review finding, or changing the meaning of a status value can make old fixtures appear to pass while downstream users interpret them differently. Version the contract in the experiment record, keep compatibility decisions explicit, and add a regression case for each newly enforced safety rule. A schema migration is an interface change, not merely a prompt edit.

Alternatives

A direct SDK has the smallest abstraction surface and makes HTTP/API behavior easy to inspect. LangChain is useful when several providers, prompt components, or structured runnables truly need common composition. A local model adapter or handwritten JSON validation may fit an offline tool better. The best choice is the one whose failures the team can understand and test.

A simple typed function around a provider SDK is often enough for one model and one task. Use an orchestration framework when common adapters, message handling, retrievers, or tracing reduce repeated, tested code. Avoid an abstraction merely to make a prototype look agentic: every layer adds defaults, version constraints, and a new failure vocabulary that tests and benchmarks must cover.

Template engines, tool-call APIs, and model-specific JSON modes are additional alternatives. Choose them when they solve a demonstrated integration problem, such as a provider's reliable native structured output, rather than because a schema looks more official. The invariant survives the implementation choice: only retrieved evidence may be cited, malformed or

unsupported output is visible, and the resulting object is a proposal awaiting human approval.

When to use it—and when not to

Use structured planning when downstream code needs named fields, auditable evidence paths, and predictable failure handling. Do not use it to disguise an unsupported conclusion as a typed object, or to bypass a human approval step. For a one-off inspected model call, a direct prompt may be clearer than adding an orchestration dependency.

Use schemas at a boundary where another program will inspect fields, route a workflow, or present a reviewable plan. Do not confuse a schema with a capability grant: a structured plan remains read-only until a separate, least-privilege action is explicitly approved.

Evidence trail

Read research/06-structured-ai-systems/notes.md, run experiments/10-systems/compare_structured_planning.py, and use benchmarks/06-structured-systems/README.md for the no-network contract run.

Takeaway

The relevant comparison is not “SDK versus framework.” It is whether both paths preserve the same evidence, expose failure, and stop before any consequential action. That contract is what lets the next chapter add state without adding unsafe autonomy.

Run the lab on your Mac

QR code for Chapter 10 lab

Online lab: https://hassanvfx.github.io/mac-ai/labs/10

Repository: experiments/10-systems/compare_structured_planning.py

```
uv run --group agents python experiments/10-systems/compare_structured_planning.py
```

Expected: Structured plan/review comparison.

Evidence: benchmarks/06-structured-systems/README.md

Scan the code to open the live lab, then use Safari Share → AirDrop to send it to your Mac. The lab follows main; clone the repository once and run the command from its root directory.

Agents Are State Machines

Intuition

An agent is easier to trust when we can name its states and legal transitions. "Think, act, observe" is a useful sketch, but a production-minded workflow needs to answer sharper questions: what state is saved, what causes a pause, what happens after rejection, and which transition is allowed to mutate the world?

State is also an audit record. It should say which evidence was considered, which plan was proposed, why a workflow paused, and whether a human accepted or rejected it. If those facts exist only in transient prompt text, a restart or a later review cannot distinguish a deliberate decision from an accidental one.

This perspective removes unnecessary mystery. A language model may help choose words in a plan, but the workflow still has inputs, state updates, branches, terminal conditions, and side effects. Naming them lets a reviewer ask ordinary questions: which field came from retrieval, which node can change it, what happens if a node is replayed, and which final states are acceptable?

Problem

The Book Intelligence Assistant has retrieved evidence, a plan, and a critique. It needs a durable approval boundary before any future writing capability. A single function can hide that boundary; a state graph makes it testable.

The graph must represent more than a happy path. It needs a no-evidence path, an interrupted path, an approval path, a rejection path, and a safe response to an unavailable model. If these outcomes live only in prose around a prompt, they are easy to forget during a refactor. If they are states and edges, a test can enter each branch and assert its terminal status without relying on a model to behave consistently.

Write an invariant beside each transition. In this graph, evidence paths are read-only inputs from retrieval; a planning node may add a proposal but cannot add authority; the approval node is the only source of a decision; and both terminal nodes are explicitly no-write. An invariant narrows review:

instead of asking whether an agent "seems safe," ask whether any edge can reach a state-changing node without an approval event scoped to the proposed action.

State is not the same thing as a transcript. Save the smallest information needed to resume and audit the workflow: objective, evidence paths, proposal, critique, model/fallback status, decision, and terminal status. Avoid placing credentials, whole repositories, or unnecessary private prompts in a checkpoint. A useful checkpoint should help recover a pending decision without creating a new unreviewed data store.

Minimal implementation

approval_workflow.py defines a small LangGraph state: objective, evidence paths, plan, critique, model status, decision, and terminal status. Its edges are explicit:X

```
START → plan → critique → approval interrupt → approved_no_write | rejected_no_write
```

The plan and critique nodes are deterministic in this first version. If no evidence is available, the plan is empty and the state records a deterministic fallback. There is no tool that edits a file or runs Git.

Treat the state type as an interface. objective identifies the proposed work; evidence_paths identifies what the proposal may rely on; plan and critique are reviewable intermediate products; model_status makes deterministic fallback visible; and approved plus status record the decision and outcome. The graph does not store an implicit "go ahead" between fields. Its conditional edge reads an explicit decision and routes only to named no-write terminal nodes. This is easier to audit than a long loop that decides for itself when to call a tool.

Before adding nodes, write the transition table in words: what causes entry, what data it reads, what it writes, whether it can be retried, and whether it has an external effect. A node with an external effect deserves a separate approval boundary, not merely an earlier confirmation in an unrelated branch.

The transition table is also a testing plan. For every conditional route, make one deterministic fixture that reaches it. For every persisted state, close and reopen the storage used by the demonstration. For every purportedly

harmless node, assert that a tracked source remains unchanged. This is why a state graph is useful even before a model is selected: its reliability properties can be verified with ordinary data and control flow rather than sampled language.

Real implementation

Run the local SQLite-backed graph:

```
uv sync --group agents
uv run --group agents python experiments/11-langgraph/approval_workflow.py
uv run --group agents python experiments/11-langgraph/approval_workflow.py \
  --approve --thread-id chapter-11-approved
```

The first invocation pauses at interrupt(). The second invocation in this small demonstration resumes immediately with the declared decision. LangGraph checkpointing ties state to a thread_id, allowing a later process to resume the same workflow (LangChain 2026c, 2026b).

The checkpoint is operational state, not published evidence. It lives under the ignored .book-intelligence/ directory so an interrupted run can resume locally without putting transient plans or decisions into Git. A thread_id names one sequence of work. Starting an unrelated proposal with an old ID risks mixing its history with a new objective, so use a new ID for a new proposal and resume only when the pending decision is genuinely the same one.

Thread IDs are part of the integrity boundary. Generate a new, descriptive ID for each independent revision request, and display the resumed objective and evidence paths before accepting a decision. If a user cannot recognize the pending work, reject it and start a fresh graph. A checkpoint proves that some state was saved; it does not prove that it still matches the current checkout. For a long-lived workflow, compare an evidence revision or content hash at resume time and route stale evidence back to retrieval rather than approving an obsolete plan.

The SQLite example is intentionally local and single-user. Its role is to demonstrate checkpoint/resume semantics, not to prescribe a shared production database. Concurrent users, hosted workers, retention requirements, backup, and access control need a separately designed store. Carry the same state contract forward, but do not assume that a file created for a tutorial is a complete operational design.

Experiment

Run the rejection and approval cases with distinct thread IDs. Confirm their terminal states are rejected_no_write and approved_no_write; inspect the SQLite file only as generated local state. Then run the test that opens a new SQLite connection after the interrupt and resumes the old thread. That is the actual persistence property demonstrated here.

Inspect the pause payload before supplying a decision. It contains the objective, evidence paths, plan, critique, and an explicit statement that no source, Git, or external action will follow. Resume once with rejection and once with approval under separate IDs. The important observation is not that a graph resumes—it is that both paths are visible, deterministic, and end without modifying a tracked source. The automated reopen test is stronger evidence than leaving one SQLite file on a laptop after a manual run.

Test the state trace, not just the final label. A rejected workflow should include the retrieved paths, an empty-or-supported plan according to its input, the critique, the interruption payload, and the rejected_no_write terminal status. An approved workflow in this beta must end at approved_no_write too: approval confirms review of a proposal, not permission for hidden execution. This distinction prevents a misleading future change where the approved branch quietly gains a file-writing tool because it already has a boolean named approved.

For an unavailable planning model, preserve the fallback status in state and continue only with the deterministic proposal policy. Do not disguise fallback text as a model result, and do not make a later resume choose a provider that was not part of the original request. A human can decide to rerun the workflow under a declared model configuration; that is a new experiment, not automatic recovery.

Worked trace: reject, reopen, and approve

Begin with a fresh thread ID such as chapter-11-reject. The graph receives an objective and the evidence path for Chapter 8. Its deterministic plan node creates three review steps; its critique node adds two checks; then the approval node emits an interrupt payload. At this point the persisted state has an objective, one evidence path, plan, critique, deterministic model status, and an explicit action message: no source, Git, or external action will follow. It has no file diff and no writer capability.

Resume that same thread with {"approved": false}. The conditional edge selects the rejected node, which records rejected_no_write. The test places a small source fixture beside its SQLite database and compares its text after the terminal state; it remains original. The lesson is not that rejection is hard to implement. It is that the rejected route is a named, durable outcome with a direct assertion against the side effect we refuse to permit.

For persistence, start a separate chapter-11-approve thread and stop at the same interrupt. Close the SQLite connection completely. Reopen the database in a new connection, rebuild the graph with its checkpointer, and resume with {"approved": true}. The terminal status becomes approved_no_write. The reopened process did not reconstruct a proposal from a chat transcript: it read the checkpoint associated with that thread ID. Yet approval still did not grant write authority, so the positive path is as safe to exercise in a test as the negative path.

Finally, invoke the graph with an empty evidence list. The plan is empty and the saved model_status becomes deterministic_fallback_no_evidence, while the graph still pauses for inspection. This trace prevents an important shortcut: a workflow cannot replace absent evidence with a confident default plan simply because it has a route to an approval screen. In a real review, the human should reject or restart with retrieval; the test asserts only the deterministic, no-model contract.

What broke

The first synchronous SQLite run failed because Python normally binds a connection to the thread that created it, while the saver can checkpoint from a worker thread. The local fix is check_same_thread=False, as recorded in benchmarks/07-workflow-graphs/README.md. The more important lesson is not the flag: persistence is a real systems dependency with concurrency behavior that must be tested rather than assumed.

Retries expose a second hazard. In an interrupt-capable graph, code before an interrupt can run again on resume. It must therefore be idempotent: constructing a plan payload is safe to repeat, while sending a message, editing a file, or charging an account is not. If a future system adds a writer, put it after a fresh, scoped approval and make its intended diff inspectable. Never hide an irreversible effect in a node that may be replayed.

Stale approval is a third hazard. A person may approve a plan, the underlying chapter may change, and a delayed process may later resume

with evidence that no longer supports the proposed edit. The beta prevents writes entirely, which keeps this case safe. A future writer must revalidate evidence and display the exact intended diff immediately before a scoped approval; a broad "continue" button is not enough.

Alternatives

For a short, stateless task, a plain function and an explicit confirmation can be clearer than a graph. An in-memory checkpointer is useful in unit tests. A database-backed checkpointer becomes worthwhile when an interrupt must survive a process boundary. Other backends may be better for concurrent, hosted, or regulated deployments, but they need their own operational design.

A conventional pull-request workflow is also a graph, implemented by people and version control: gather evidence, prepare a diff, review, approve, merge or reject. For many editorial tasks it is clearer than an agent runtime. Use a programmatic graph when durable routing and interruption reduce real work, not because state diagrams make a simple question more impressive.

Event logs or a conventional job queue can also model parts of this workflow. They are useful when many independent tasks need scheduling, but they do not replace an explicit approval state and evidence contract. A graph library earns its complexity when its persisted routing, interrupt semantics, and testable state are requirements; otherwise a typed function plus a reviewable pull request remains a strong, lower-risk alternative.

When to use it—and when not to

Use a graph when state, recovery, routing, or human control is part of the product behavior. Do not use a graph merely to make one model call look more agentic. A graph does not create safety by itself; its nodes must still have least privilege.

Use state graphs when a user needs to inspect progress across a process restart, when approval and rejection must route differently, or when checks must run in a declared order. Do not use them to conceal broad tool access behind a chat interface. A state machine can make unsafe authority easier to automate as well as safe authority easier to test.

Evidence trail

The workflow source note is research/07-workflow-graphs/notes.md. Run experiments/11-langgraph/approval_workflow.py and review the local-persistence observation in benchmarks/07-workflow-graphs/README.md.

Takeaway

An agent is a state machine with a language model somewhere inside or beside it. Defining the state first makes interruption, recovery, and permission boundaries concrete instead of aspirational.

Run the lab on your Mac

QR code for Chapter 11 lab

Online lab: https://hassanvfx.github.io/mac-ai/labs/11

Repository: experiments/11-langgraph/approval_workflow.py

```
uv run --group agents python experiments/11-langgraph/approval_workflow.py
```

Expected: A workflow that stops for approval.

Evidence: benchmarks/07-workflow-graphs/README.md

Scan the code to open the live lab, then use Safari Share → AirDrop to send it to your Mac. The lab follows main; clone the repository once and run the command from its root directory.

Memory, State, and Human Control

Intuition

"Memory" is overloaded. A model's learned weights, retrieved documents, a chat history, and a database checkpoint are different things with different failure modes. For a safe workflow, the relevant memory is saved state that a person can inspect before deciding whether to continue.

Approval must be scoped to a specific, reviewable proposal. It is not standing permission for an agent to edit unrelated files, rerun a different plan, or call an external service later. Binding the decision to recorded evidence and the proposed change makes the boundary meaningful when state is resumed.

Problem

A chapter-revision proposal may take longer than one process or one work session. The assistant must preserve its plan and critique without treating a paused workflow as implicit approval. It must also distinguish a durable checkpoint from the repository itself: the checkpoint is local operational state, not canonical editorial evidence.

There are two tempting but unsafe shortcuts. The first is to keep all context only in a chat window. That makes a restart indistinguishable from a new request and gives neither the reviewer nor the program a stable record of what was considered. The second is to put the proposed edit in the checkpoint and apply it automatically after a positive response. That turns a review decision into a write capability. This book uses a narrower contract: a checkpoint can remember a proposal, but it cannot be a substitute for the repository, a permission system, or a change log.

Approval is strongest when the reviewer can answer four questions without reconstructing hidden context: what is proposed, which evidence supports it, what exact effect would follow, and what happens if the answer is no. The beta answers the third question with "nothing is executed." That may feel limited, but it is a valuable capability boundary: a reviewer can exercise approval and rejection paths before any code has authority to write prose, change Git state, or contact an external service.

Human control is also more than a boolean. A useful decision can be approve, reject, request a revision, or abandon a stale proposal. This small graph uses approve and reject to keep the state space testable. If a revision option is added later, it should route back to evidence and planning with an explicit reason, not mutate the old plan in place and reuse an approval that was scoped to different text.

Minimal implementation

The graph from Chapter 11 stores a checkpoint after each super-step under an explicit thread_id. The approval node emits a JSON-serializable payload with the objective, evidence paths, plan, critique, and a statement that no action will occur. It waits for a value passed by Command(resume=...) (LangChain 2026b).

The resumed value is only a boolean decision. Even true reaches approved_no_write, not a writer node. This deliberately separates *approval of a proposal* from *execution of a change*.

This also makes a negative outcome useful rather than exceptional. A rejection is durable information: the proposal was reviewed and is not authorized to continue. A later attempt needs a new proposal with new evidence, or a clearly identified resume of the same proposal. It must not silently reinterpret an old approval after the chapter, corpus, or requested objective has changed.

The pause payload is an interface for a human, so design it for review rather than for a model. Keep paths readable, state why each item was retrieved, show the critique and unsupported-claim warnings, and name the no-write boundary in plain language. Do not make a reviewer infer scope from a long free-form prompt. A good interface makes it easy to reject an unclear proposal; rejection is a safe outcome, not an error that must be optimized away.

Real implementation

The local example uses SqliteSaver because a file makes resume testable after the original connection closes. Its path is .book-intelligence/approval-workflow.sqlite, which is ignored by Git. The test suite proves that a source fixture remains unchanged after rejection and that a reopened SQLite store can complete a previously interrupted approved thread.

Use one new thread_id for each independent proposal. Reusing a thread ID is a resume operation, not a clean new request. That is why stable IDs should be chosen deliberately and never derived from secret or sensitive user content.

The checkpoint has a limited retention purpose. It records workflow state such as the objective, paths used as evidence, plan, critique, decision, and terminal status. It should not become a second index of the whole book or a dumping ground for retrieved documents. Keeping evidence as repository paths means a reviewer can reopen the canonical file, while keeping the local SQLite file ignored prevents operational state from polluting a commit or an exported manuscript.

Retention is a design choice, not a side effect of choosing SQLite. Decide how long pending proposals are useful, who may read their objectives and evidence paths, where backups reside, and how generated state is removed when the work ends. For this repository, local generated state is ignored and can be deleted without deleting canonical prose or experiment records. A shared deployment would need access controls and an explicit retention/deletion policy before storing project context for more than a local learning exercise.

Do not put raw secrets in an approval payload, even if a future action needs an API credential. The credential should be resolved only by the least-privileged executor after a scoped approval, and its value should never be checkpointed or rendered in logs. The current graph has no executor, so it offers a simple property to test: there is nowhere for a secret-backed external call to occur.

Experiment

Pause the graph, close the process, reopen the SQLite checkpoint, and resume with rejection. Check that the terminal state is rejected_no_write and no source changed. Repeat with approval and verify that it still ends in approved_no_write. These two outcomes make the human boundary observable.

The runnable implementation is experiments/11-langgraph/approval_workflow.py; its durable behavior is checked in tests/test_approval_workflow.py. The tests create an isolated SQLite file, invoke until the interrupt, reopen the database in a new connection, and resume it. They also start a graph with no evidence and

verify the deterministic fallback. That last case matters: an unavailable model or empty retrieval result must reduce the proposal's authority, not encourage the system to invent support.

For a manual inspection, copy the experiment's local checkpoint path to a temporary directory, run it once until it reports an interrupt, then run the resume command with {"approved": false}. Inspect the resulting state and confirm that no tracked source file has changed. Repeat in a fresh thread with approval. The expected terminal labels are deliberately explicit so logs and tests do not mistake a pause for success.

Run a stale-proposal exercise as well. Pause with one objective and evidence set, change or replace a cited file in a disposable fixture, then inspect the resume boundary. The safe policy is to invalidate or re-review the proposal, not to treat the old decision as portable. The current beta expresses this as a design rule because it has no writer; a writer-capable version must enforce it with a revision/hash check and a new approval payload containing the intended diff.

Review ergonomics are measurable. Record whether a reviewer can identify the objective, paths, limitations, decision, and final no-write status from the payload and trace. If they cannot, the system has a transparency failure even when the graph reaches the expected node. A future usability evaluation can use scripted review cases, but it should preserve the same safety rule: a positive label is not evidence of an executed or correct change.

Worked scenario: a proposal becomes stale

Imagine a reviewer pauses a proposal to improve the embeddings chapter. The checkpoint names book/chapters/08-turning-meaning-into-geometry.md, stores a plan and critique, and asks only whether the proposal should proceed to its current beta terminal state. While the workflow is paused, another editor rewrites the retrieval experiment and changes the chapter's evidence trail. The stored path still resolves, but the evidence and intended revision are no longer necessarily the same work the reviewer first saw.

The safe response is not "approved once, therefore write now." First compare the current evidence revision or content hash with the one that informed the proposal. If it differs, record the proposal as stale; retain its local history for the configured retention period; and start a new retrieval/plan cycle with a new thread ID. The old approval or rejection

remains a fact about the old proposal, not a permission or prohibition that automatically transfers to a different manuscript state.

Suppose instead the reviewer rejects because the plan lacks a benchmark record. The checkpoint ends rejected_no_write; no source is changed and no ambiguous "pending" state remains. A later author can add the missing evidence, but that creates a new objective and must produce a new proposal. The original rejected thread remains useful as an audit clue, while the new thread avoids silently erasing why the earlier plan was declined.

Only after a future writer has re-read the current evidence should it calculate one proposed diff. The reviewer must see that diff, its target file, the evidence paths, and the expected test commands in a second scoped approval. Even then the writer should receive only that target and diff—not unrestricted repository access. This two-approval shape is intentionally stricter than the beta graph, whose useful demonstration is that both approval and rejection are durable while still incapable of changing a tracked file.

What broke

An interrupt node restarts from the beginning when resumed. Therefore code before interrupt() must be idempotent and must not perform an irreversible action. This graph constructs only a JSON payload before the pause. If a future writer is added, it must be a separate post-approval node with its own explicit confirmation and tests—not an accidental side effect of rebuilding the approval payload.

Another failure is treating the checkpoint as current evidence. A path can move, a benchmark can be superseded, and a chapter can change while a thread is paused. Before any future writer acts, it must re-read every cited path, validate that the proposal still applies, and show the resulting diff to the human. A saved approved flag alone is never enough to prove that a changed proposal remains safe.

Rejection needs an equally clear recovery route. Preserve the rejected status and reviewer reason when there is one, then require a fresh retrieval and plan for materially changed work. Do not reset a rejected checkpoint to pending in place, because that erases the audit trail and can confuse a later reviewer about which proposal was actually declined. Starting a new thread is cheaper than interpreting a history with ambiguous scope.

Alternatives

A manual ticket, pull request, or command-line confirmation can be the right human-control mechanism for a simpler workflow. Stateless requests avoid checkpoint retention but lose recovery. Hosted database checkpointing may suit teams, while local SQLite keeps a single-developer learning project easy to inspect. None of these replaces access control or a clear retention policy.

It is also reasonable to avoid an agent graph altogether. A deterministic script that gathers links and runs the editorial audit is easier to reproduce and often sufficient. Use a persisted graph only when resumable, inspectable workflow state provides a real benefit. The comparison in Chapter 13 treats that extra machinery as a cost to be justified by the task, rather than a default sign of sophistication.

When to use it—and when not to

Use persisted state when interruption and recovery are requirements. Do not persist more context than the next decision needs, and do not put credentials, private journals, or unreviewed generated text into a checkpoint by default. If there is no safe execution path yet, the correct terminal state is still a proposal, not an automatic change.

For collaborative work, combine the checkpoint with ordinary version control: the graph identifies a reviewed proposal; a human creates or approves the actual diff; Git records the final change. This division is intentionally boring. It gives a reader one place to inspect operational history and another place to inspect published source history, without asking either system to pretend it can replace the other.

A future writer should be a narrow, separate capability: receive an approved proposal and freshly verified evidence; calculate a single inspectable diff; pause again with that diff; then write only the approved target if the revision still matches. It should not receive a shell, a broad filesystem path, Git credentials, or network tools by default. This is more restrictive than many agent demonstrations, but it turns “human in the loop” into a technical property instead of an honor system.

Evidence trail

Read research/07-workflow-graphs/notes.md, run the no-write graph at experiments/11-langgraph/approval_workflow.py, and inspect tests/test_approval_workflow.py for persistence, rejection, and fallback contracts.

Takeaway

Human-in-the-loop is not a decorative confirmation button. It is a persisted state transition whose resume behavior, data scope, and failure paths must be designed before the system earns the right to act.

Run the lab on your Mac

QR code for Chapter 12 lab

Online lab: https://hassanvfx.github.io/mac-ai/labs/12

Repository: experiments/11-langgraph/approval_workflow.py

```
uv run --group agents python experiments/11-langgraph/approval_workflow.py --approve
```

Expected: Persisted approval and resume behavior.

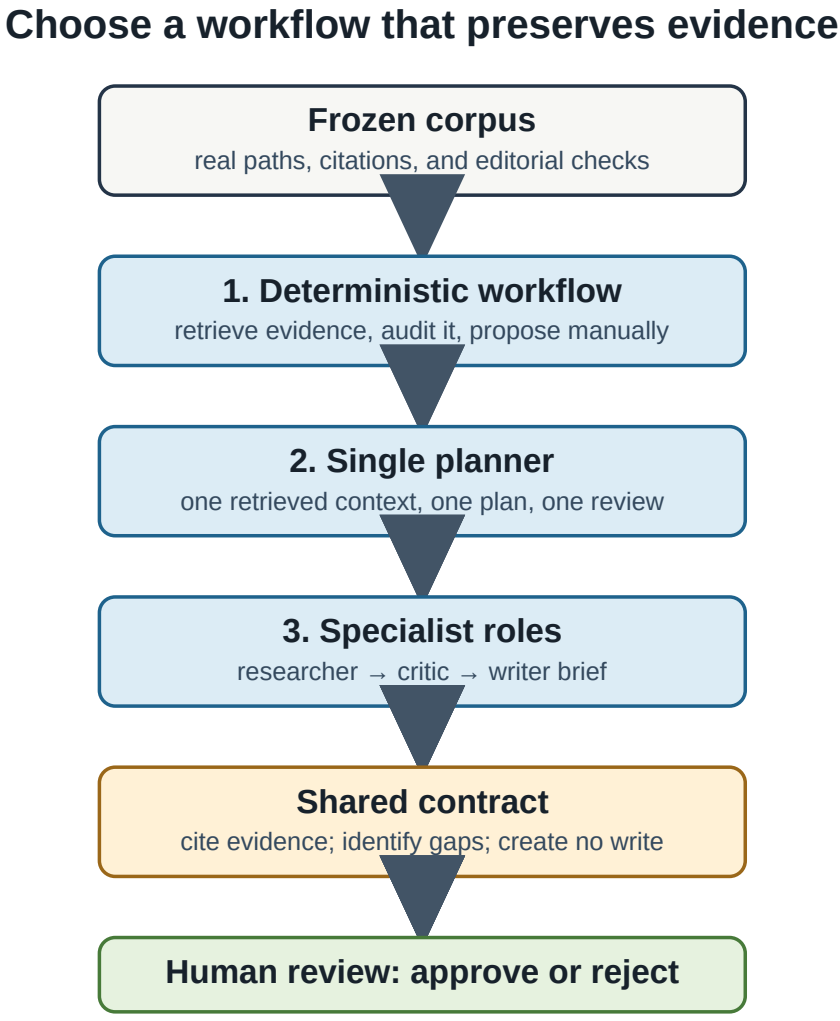
Evidence: benchmarks/07-workflow-graphs/README.md

Scan the code to open the live lab, then use Safari Share → AirDrop to send it to your Mac. The lab follows main; clone the repository once and run the command from its root directory.

When One Agent Is Not Enough

Intuition

Multiple agents add specialization and review, but also coordination cost and new failure modes. The useful question is not how many role labels a workflow has; it is whether the added handoff catches a real risk.



Three workflow shapes converge on the same evidence and approval contract.

The diagram is deliberately asymmetric. The three routes have different internal steps, but they enter the same shared contract before the human sees a proposal. A system does not become safer merely because it has a researcher, a critic, and a writer in its prompt. It becomes safer only when each handoff has a visible input, an inspectable output, and a testable boundary on what can happen next.

Problem

Compare three safe shapes on the same frozen book-maintenance task: deterministic retrieval and review, one planner, and researcher/critic/writer roles. Every shape must cite the same evidence, flag the same missing editorial section, require human approval, and make no write.

This is a contract comparison, not a model leaderboard. The fixture asks a concrete maintenance question: locate the explanation of cosine similarity and review a deliberately incomplete Chapter 8 draft. The expected evidence path and missing Alternatives section are versioned with the fixture. That lets the comparison fail if a role loses provenance, skips the review, or acquires write behavior, even when no remote model, API key, or nondeterministic output is available.

Choose the task before choosing the roles. The task should have a known evidence path, a reviewable editorial condition, and an observable safe outcome. A role count is not a treatment variable by itself: a three-role graph that changes retrieval, prompts, and evaluation at the same time cannot tell us why a result changed. The frozen fixture holds those variables still so the first question is narrow: do all shapes preserve the same basic contract?

There are at least three costs to count. Every handoff can add latency and model calls; every copied context can lose a source path or qualification; and every role-specific tool can broaden authority. A specialist is justified only when its distinct input and output prevent or expose a failure the smaller workflow misses. “Researcher,” “critic,” and “writer” are responsibilities to test, not personality prompts that automatically create independent judgment.

Minimal implementation

workflow_comparison.py keeps the roles deterministic so the comparison is repeatable without an API key. The writer produces only a brief; it has no file or Git capability.X

The control path calls the ordinary retrieval and corpus-review functions and leaves interpretation to a human. It is the baseline for both complexity and reliability: if it already finds the relevant path and editorial omission, a more elaborate graph must demonstrate what it adds. The single-planner path uses one retrieved context to make a short plan. The specialist path turns the same work into explicit handoffs, making each role's responsibility visible in the report rather than hiding it inside a long prompt.

The reports are a comparison interface. evidence_paths answers what each shape may cite; plan_steps reveals the proposed work; findings records the mechanical review; writer_brief shows whether a drafting role stayed within its remit; and the two booleans state the approval and no-write invariants. This is deliberately less dramatic than a chat transcript. It lets a test and a human compare the same fields across shapes without inferring safety from the tone of a generated explanation.

The writer role is a particularly useful naming test. In the current code it has no filesystem, Git, shell, or network capability and emits only a brief. If an implementation cannot state this boundary in its report and tests, it is not a safe writer role for the beta. Future prose-editing assistance belongs behind the separate, diff-scoped approval design from Chapter 12—not behind an ambiguous role name.

```
uv run --group agents python experiments/13-workflows/compare_workflows.py
```

Real implementation

The deterministic workflow is the control. The single planner combines retrieval, planning, and review context. The role pipeline separates researcher, critic, and writer responsibilities, then stops for approval. All three passed the same path-attribution, review-coverage, approval-boundary, and no-write checks in the frozen fixture. See benchmarks/08-workflow-comparison/README.md.

Each WorkflowReport carries the evidence paths, ordered plan steps, findings, writer brief, approval requirement, and an explicit

writes_performed flag. Those fields are intentionally mundane. They give tests something more useful than a persuasive paragraph to inspect. In particular, the researcher/critic/ writer variant cannot hide a write behind its writer label: the writer produces a brief, and the report records that no write occurred.

An API-backed version may replace a deterministic role's text generation, but it must preserve this envelope. The provider configuration belongs in process environment variables, the retrieved paths must remain attached to every proposal, and malformed structured output must fail closed. Until the project records a controlled evaluation with a selected model, latency, cost, answer quality, and failure recovery are unknown—not evidence for choosing one shape.

An API-backed extension should preserve a per-role trace: task ID, corpus revision, retrieved paths, prompt/schema version, model configuration, start/end times, parse result, and final contract checks. It should not persist credentials or raw private context in that trace. This makes a slow or failed run interpretable: was the extra delay retrieval, a critic request, parsing, or human interruption? Without a trace, total elapsed time encourages teams to blame or credit the role arrangement for unrelated provider behavior.

Experiment

The recorded output shows all four contract checks as true. This does not show that a multi-agent system is more accurate or useful: no remote model was run. It establishes a baseline that an API-backed comparison must beat under a declared quality, cost, and latency protocol.

Run the comparison, then inspect its compact table rather than treating a passing exit code as the whole result. Every row should attribute the fixture's expected research-note path, report the missing-alternatives finding, require approval, and report no write. A failed cell identifies the broken contract directly. The accompanying test repeats the same assertions in tests/test_workflow_comparison.py, so a refactor cannot silently widen a role's authority.

For a future controlled comparison, hold the corpus, objective, retrieval count, model settings, and evaluator fixed. Pre-register a small scoring rubric for citation accuracy, plan completeness, review usefulness, latency, and unsafe-action refusal. Record failures as well as successful traces. Only then

ask whether an extra critic catches omissions that a single planner misses often enough to justify its additional requests and coordination.

Define success before looking at the outputs. Citation accuracy can require every displayed path to come from the retrieved set; plan completeness can be scored against named required sections; review usefulness can ask whether a known omission was surfaced without inventing one; unsafe-action refusal must remain perfect for this beta. Measure latency and request count separately from quality. A workflow that finds one more issue but takes ten times as long may still be the wrong default for a routine editorial check.

Use paired cases where possible: run every shape on the same maintenance question and preserve every refusal, timeout, malformed response, and reviewer disagreement. Do not average failures out of an appealing summary. The small fixture is not enough to estimate general performance, but it establishes how to retain comparable traces before an optional model experiment is authorized.

Worked case: an incomplete embeddings chapter

The frozen fixture asks: "Explain cosine similarity and review the draft chapter." Its small research file, evals/fixture_corpus/research/embeddings.md, contains the expected cosine-similarity evidence and citation key; its deliberately incomplete evals/fixture_corpus/book/chapters/08-draft.md lacks an Alternatives heading. That gives the comparison two independent checks: retrieval must preserve the research path, and review must flag the editorial omission. It does not ask any workflow to judge whether a large real chapter is well written.

Run the three shapes and read the resulting table as follows. The deterministic control returns four generic, evidence-bound proposal steps and the mechanical missing-section finding. The single planner returns three compact steps after the same retrieval. The researcher/critic/writer path returns four named steps: verify paths and citation keys; identify unsupported claims or omissions; prepare a proposal only; then wait for a person. In the recorded no-network run, every row passes path attribution, review coverage, approval boundary, and no-write checks. The difference is workflow structure, not a demonstrated difference in model quality.

Suppose the role pipeline lost the fixture path evals/fixture_corpus/research/embeddings.md at its writer handoff while the researcher retained it. The path_attribution cell would fail even if the writer's prose sounded convincing. Suppose all roles found the path but the critic failed to inspect the draft heading. The review_coverage cell would fail. Suppose a future refactor added file output to the writer. The no_write cell would fail. This is why the table contains separate cells: one green result cannot conceal a different broken property.

The appropriate conclusion is modest. For this fixture, the specialist shape adds an explicit critique and writer brief while preserving the same contract. It has not shown a quality benefit large enough to justify model calls on a larger corpus. The control remains the default until a pre-registered, model-backed evaluation demonstrates a task-specific advantage under the latency, cost, and safety criteria already stated.

What broke

Role separation can duplicate retrieval, lose context at a handoff, or make responsibility less clear. A writer role is especially dangerous when it can quietly turn a proposal into a mutation. This implementation avoids that risk by making its output a read-only brief.

Another common failure is role theatre: several agents repeat the same search, then agree with one another because they share the same incomplete context. A critic is useful only if it receives an independently inspectable target—such as the plan, cited paths, and editorial checklist—and can return a finding that the workflow records. If a handoff has no distinct input or decision, remove it. It adds latency and a new place for evidence to disappear without adding a testable safeguard.

Specialization can also create correlated errors. If all roles receive the same bad retrieval result, a researcher, critic, and writer may repeat the same unsupported premise with three different voices. Independence requires a different check, such as a deterministic path audit, an alternate retrieval query, or a reviewer who can reject the premise—not merely separate prompt labels. Keep the evidence packet visible at every handoff so agreement can be examined rather than mistaken for corroboration.

Alternatives

Use one deterministic workflow for routine, inspectable checks. Use a single planner when one coherent proposal is enough. Add specialist roles only when their independently testable review adds value. A pull request or human editor can be a better critic than another model.

A fixed rules engine is often the best first critic for manuscript work: it can check headings, paths, citation keys, and missing required sections exactly. Use a language model only for the residual work that needs interpretation, such as suggesting an experiment or identifying a confusing explanation. This division keeps objective checks reproducible and makes the model's uncertain judgment explicit.

Another alternative is a two-stage workflow: deterministic retrieval and audit first, then one optional model-assisted explanation only when the first stage found supported evidence. This can capture much of the useful editorial help without paying the coordination cost of a full graph. Escalate to specialist roles only after a versioned failure set shows which distinct check is missing.

When to use it—and when not to

Use multi-role workflow when research, criticism, and drafting have different evidence needs and their handoffs are observable. Do not use it to manufacture confidence, parallelize an already simple task, or evade approval.

Choose roles from the failure you need to reduce. If retrieval occasionally loses citation keys, strengthen retrieval tests before adding a writer. If a plan routinely omits alternatives, a deterministic editorial audit may be cheaper than a critic model. If a high-stakes revision needs separate evidence review and explanatory drafting, then a role boundary can make review easier. In every case, a human remains responsible for deciding whether the proposed change should become a repository change.

Evidence trail

Read research/07-workflow-graphs/notes.md, run experiments/13-workflows/compare_workflows.py, and inspect the frozen comparison contract in benchmarks/08-workflow-comparison/README.md.

Takeaway

More agents are justified only by measured, auditable improvement. Until then, the smallest workflow that preserves evidence and human control is the best one.

Run the lab on your Mac

QR code for Chapter 13 lab

Online lab: https://hassanvfx.github.io/mac-ai/labs/13

Repository: experiments/13-workflows/compare_workflows.py

```
uv run --group agents python experiments/13-workflows/compare_workflows.py
```

Expected: Deterministic workflow comparison.

Evidence: benchmarks/08-workflow-comparison/README.md

Scan the code to open the live lab, then use Safari Share → AirDrop to send it to your Mac. The lab follows main; clone the repository once and run the command from its root directory.

Building an AI System You Can Actually Trust

Intuition

Reliability comes from evaluation, observability, constraints, and recovery—not from a single impressive demo. A fluent plan is not reliable unless its evidence, failure mode, permission boundary, and trace can be inspected.

For this book, "trust" is deliberately narrow. It does not mean that the assistant knows whether a technical claim is true, that a retrieved paragraph is complete, or that a proposed revision is good prose. It means the program can demonstrate a few concrete behaviors: preserve an evidence path, preserve a citation key when one is present, refuse when evidence is missing, keep a proposal separate from a write, and leave an inspectable record of those checks. Narrow promises are more useful than a broad promise that no test can prove.

Problem

The Book Intelligence Assistant combines retrieval, planning, critique, and approval. The beta needs a repeatable way to prove its narrow guarantees and state plainly what it cannot yet guarantee.

Reliability has to survive ordinary changes. A new chapter can alter retrieval ranking; a refactor can drop a citation key; an optional model installation can be unavailable on a reader's machine; a graph resume can be mistaken for an authorization to act. A one-off demo will usually miss these regressions. The project therefore treats a versioned fixture corpus and deterministic evaluator as a safety net, while keeping the much larger live book corpus local for interactive work.

Start a reliability policy with an inventory of promises. For each promise, name the input boundary, the code path that enforces it, a deterministic test case, the trace field that makes the result inspectable, and the condition that would make a release fail. "The assistant is safe" is not an inventory entry. "A plan cannot retain an evidence path outside its retrieved allow-list" is. The narrower wording tells a contributor where to look when a regression appears and tells a reader exactly what has—not—been demonstrated.

Reliability also requires separation of concerns. Correct mathematical experiments, accurate editorial prose, grounded retrieval, valid navigation, and print-ready layout are related release properties but have different authoritative checks. A green Python test suite cannot prove a diagram prints legibly; a successful DOCX conversion cannot prove a generated plan cites the right source. The beta release gate must assemble evidence from each layer without letting one convenient green command stand in for all of them.

Minimal implementation

The frozen fixture corpus and reliability runner exercise evidence location, missing-evidence refusal, no-write planning, and editorial review. It writes a JSON trace only under ignored .book-intelligence/, with a case count, result details, and explicit no-write policy.X

The frozen cases are small on purpose. They test six distinct contracts:

locating a known evidence path;

returning a grounded answer with that path visible;

carrying a BibTeX citation key from indexed research;

refusing an unsupported ISBN question;

proposing an experiment without modifying a source; and

flagging a deliberately missing editorial Alternatives section.

The trace is generated local state, not book evidence and not a substitute for test output. Its safe location makes it useful for inspecting a run without asking contributors to commit potentially noisy details. A release record can summarize its result, but should link to the versioned fixture and evaluator that made the trace meaningful.

The fixture cases are specification examples, not merely regression data. A new capability should arrive with at least one positive case, one missing or malformed-input case, one provenance or authority-boundary case, and a clear expected outcome. Keep cases small enough that a failing result points to a specific contract. If a case needs a real provider, split its deterministic structural checks from its explicitly configured, non-secret benchmark instead of making ordinary contributors depend on a credential.

```
uv run --group agents python evals/run_reliability.py
```

Real implementation

The system's reliability policy is layered. Retrieval must retain repository paths and citation keys. Grounded answers return evidence or refuse. Structured plans discard invented paths. LangGraph pauses before approval and its approved terminal state still performs no write. These controls are regression-tested without a credential or remote model.

Think of the controls as a chain, not a single guardrail. Ingestion attaches provenance to each chunk. Retrieval passes that provenance to the answer or plan layer. The answer layer refuses to manufacture support when the retrieved set is empty. Planning filters paths against the repository evidence it was given. Review checks deterministic editorial requirements. Finally, the graph records an approval decision but reaches an explicit no-write terminal state. If one link is removed, a later layer must not guess that the earlier guarantee still holds.

The policy also separates deterministic and model-dependent behavior. Fixture retrieval, path validation, source-boundary checks, required-section review, and approval-state transitions should pass without network access. A language model may help phrase a plan or critique, but it inherits the same evidence and approval requirements. Missing API configuration, unavailable local weights, or malformed structured output are failures to surface, not invitations to fall back to an uncited free-form answer.

Define graceful degradation before an outage occurs. If learned embeddings are unavailable, label and use the deterministic retrieval baseline for contract tests; do not claim semantic search occurred. If a model adapter is unavailable, return its configuration error or continue only through an explicitly deterministic no-model route. If no evidence is retrieved, refuse rather than write a generic plan. A reliable fallback preserves the important boundary; it does not merely keep a user interface talking.

Traces should be actionable but proportionate. Include the case ID, corpus or fixture revision, declared policy, pass/fail result, and details needed to reproduce a failure. Redact secrets and avoid storing unnecessary private text. For a controlled model experiment, add provider/model identity and timing fields only after choosing a configuration. Trace retention then

becomes an operational decision with ownership and deletion rules, not an accidental pile of prompts and outputs.

Experiment

Run the reliability suite from a clean environment and inspect the generated trace. A passing trace shows that the frozen cases met their declared contract; it does not prove correctness on every future manuscript or API response. Record latency, tokens, provider/model identity, and failures only when a controlled API experiment actually runs.

Start with uv sync --group dev, then run the suite. A clean result reports the number of cases and writes evaluation.json plus reliability.json below the ignored local directory. Read a failed case before rerunning it: its detail contains the retrieved paths, refusal text, or review findings needed to decide whether the regression is in fixture data, the evaluator, or the assistant.

For a model-backed experiment, declare the corpus revision, task set, provider, model revision, prompt/schema, retries, temperature, timing boundaries, and redaction rules in a committed benchmark record. Include failed or refused requests. A median latency without request count, warm-up policy, or model identity is not a useful reliability result; it is merely a number.

Turn these observations into release gates. For a local beta candidate: install from the locked environment; run format/lint and deterministic tests; execute the fixture reliability suite; audit manuscript links, citations, and required sections; build the site; and build the DOCX. Each failure blocks the candidate until its cause and retest are recorded. The final Word-exported PDF, full page-by-page visual inspection, and Lulu-specific preflight remain separate print gates because they depend on an externally produced artifact and current publication settings.

An explicit unknown is a valid release result. This project has no selected API provider/model benchmark, no measured API latency or quality comparison, and no final Word PDF in this repository. Those absences must appear in the beta record rather than becoming blank cells in a performance table. The deterministic suite shows its limited contracts; it does not fill the missing external evidence.

What broke

The project has already observed weak retrieval neighbors, absent API configuration, malformed structured output, empty evidence, rejected approval, and a SQLite thread constraint. Each became a test or documented limitation. The dangerous failure is silent degradation: turning a missing model, source, or approval into a plausible write.

There are several subtle ways to make this worse. A trace can claim success while hiding a failed sub-step; the runner therefore retains per-case details. An evaluator can test only a mocked answer and miss broken indexing; the frozen cases build the index before checking results. A safety test can pass while a different code path writes a file; the planning case snapshots fixture files before and after the proposal. These are not proofs of universal safety, but they turn known failure modes into regressions that a contributor can reproduce.

Beware of metric theater as well as silent failures. Counting many test cases does not help if they all exercise the same happy path. A high retrieval score does not demonstrate citation entailment. A low average latency does not show that refusals are safe, that every trace was retained, or that a tail latency is acceptable. Pair quantitative observations with the contract they are meant to support, preserve failure samples, and revise the evaluation set when a real incident reveals a missing category.

Alternatives

Manual editorial review, pull requests, and conventional search remain strong alternatives. Hosted tracing can help a team, but creates privacy, retention, and vendor concerns. A small local trace is preferable while this beta has no remote model run.

Static tools also belong in the reliability stack. The book audit validates chapter sections, citation keys, and local links. The Python tests validate numerical lessons, retrieval contracts, structured-output failures, and graph states. The Docusaurus build catches broken course navigation, while the DOCX build exposes print-asset conversion problems. These tools do different jobs; none makes the others redundant.

Release checklists and human review are another alternative to automated evaluation, not an afterthought. They are especially important for claims that need subject-matter judgment, for print layout, and for changes to the safety

policy itself. Automation makes repeatable facts cheap to recheck; it does not transfer editorial, legal, or publication responsibility to a script.

When to use it—and when not to

Use the assistant to locate evidence and prepare proposals. Do not use it as an authority for unsupported claims, autonomous edits, legal/safety decisions, or external actions. Any future writer must be a separate, explicitly approved, least-privilege capability.

Use the trace to answer a modest operational question: did this revision still meet the contracts we have chosen to enforce? Do not use it to answer a much larger epistemic question: is the whole manuscript correct? A human editor, subject-matter review, and reproducible experiments remain necessary. When a new capability is proposed, define its failure cases and the evidence required for release before wiring it into a workflow with more authority.

Before increasing authority, use a capability review: What new action becomes possible? What exact evidence and approval scope are required? Which test proves rejection, unavailability, and retry behavior? What state is retained, where, and for how long? How does the feature roll back without removing source work? If these questions cannot be answered, keep the capability read-only. This provides a concrete stopping rule for beta enthusiasm.

Worked release gate: what a local beta can honestly claim

Start at a clean checkout and install the locked Python environment. Run the numerical and assistant tests, then run make audit-book. A successful result establishes only the contracts those commands cover: deterministic experiment expectations, retrieved-path and citation-key preservation, missing-evidence refusal, no-write planning, workflow state transitions, required chapter headings, valid citation keys, and resolvable local manuscript paths. Preserve the command output or CI result with the commit identifier; do not summarize it as “the book is correct.”

Next build the Docusaurus site from the canonical chapters. That result checks the presentation shell and navigation build, not whether a reader understands the material. Build the DOCX from the committed Lulu reference template. That checks the manuscript conversion and print-asset

pipeline, not final print approval. Render the DOCX to page images and inspect the pages at the intended 6×9 scale; a table break, clipped code listing, or unreadable diagram can pass every preceding text and Python check.

At this point the project can call itself a *local beta candidate* only if the word target, editorial matrix, automated gates, and interim visual proof all meet their stated criteria. It cannot yet claim a release-ready Lulu interior: the final export must come from Microsoft Word on macOS, its PDF needs preflight and page-by-page review, and current Lulu requirements must be checked against the actual upload artifact. It also cannot claim a deployed course without a chosen GitHub remote and Pages configuration.

Finally, keep the release record honest about exclusions. This repository has not run a configured API comparison, so it has no comparative API latency, token-cost, or quality result. The assigned ISBN is recorded in the print metadata and interior, but it has no final cover because page count is not frozen and no proof-order decision. These are not failed tests; they are deliberate user-gated production decisions. Naming them prevents a green local suite from being mistaken for authorization to publish or spend money.

Evidence trail

Read research/07-workflow-graphs/notes.md, run evals/run_reliability.py, and inspect the versioned cases in evals/book_intelligence.jsonl; generated traces remain local under .book-intelligence/.

Takeaway

Trust is a continuing engineering practice. This beta is trustworthy only in the narrow ways its tests and traces demonstrate—and it remains honest about the work required before production or print release.

Run the lab on your Mac

QR code for Chapter 14 lab

Online lab: https://hassanvfx.github.io/mac-ai/labs/14

Repository: evals/run_book_intelligence.py

```
uv run python evals/run_book_intelligence.py
```

Expected: Versioned reliability evaluation.

Evidence: benchmarks/08-workflow-comparison/README.md

Scan the code to open the live lab, then use Safari Share → AirDrop to send it to your Mac. The lab follows main; clone the repository once and run the command from its root directory.

Generative AI Lab

The earlier chapters built the pieces of an AI system: representations, retrieval, structured outputs, state, evaluation, and human control. This final lab asks a more practical question: what do those pieces look like when the output is creative work that a person may publish?

The answer is not a universal "generate" button. It is a creative system that turns a person's raw material into more useful options without taking away the person's voice, accounts, or final judgment. The useful pattern is a clear source boundary, narrow model responsibilities, inspectable intermediate work, deterministic checks where they are possible, and a human decision before a result leaves the project.

Intuition

Generative systems become valuable when they compress the repetitive parts of a real production process while keeping authorship visible. A lyric arrangement can move from a raw draft to performance-ready directions without losing the writer's words. A news-inspired video can move from fresh reporting context to an original media package without confusing source material with permission to reuse it. In both cases, the creator remains the owner of the voice, accounts, and final judgment.

This chapter studies two open-source projects maintained outside this book. They are case studies, not dependencies of mac-ai: Lyrics Refiner is a local arrangement workbench for Spanish-language lyrics, while Newsmusic is a staged pipeline from news context to original, review-gated YouTube-ready media.

Problem

A single prompt can hide too many decisions. "Turn these lyrics into a song" can silently replace wording. "Turn this headline into a video" can blur the boundary between context, generated media, rights, facts, credentials, and a publish action. A plausible result is not evidence that the workflow respected those boundaries.

The design problem is therefore broader than model quality: preserve the source, make each meaningful transformation visible, measure a concrete

invariant when one exists, and require a person to approve the action that changes the outside world.

Minimal implementation

A minimal generative workflow can be expressed as a small contract:

```
source → bounded model task → inspect result → deterministic check → human decision
```

The source stays identifiable. Each model task has one job. A program checks what it can check without asking the model to grade itself. The final human decides whether to retry, revise, export, or publish.

Run this chapter's companion guide before cloning either external project:

```
uv run python experiments/15-generative-ai-lab/case_study_check.py
```

It prints the official project URLs, the safe-first commands, and the conditions that must be true before a credential or publish action is considered.

Real implementation: two different production boundaries

Case study 1: Lyrics Refiner gives a writer a controllable studio

Lyrics Refiner is a local React creative studio for Spanish lyrics, including Regional Mexican styles. A writer starts with the actual lyric—not a generic prompt—and receives a structured path toward a performance-ready arrangement: style and phonetic analysis, structure, optional reference-shape matching, semantic repair, performance annotations, and ad-libs. The payoff is leverage without a black box: the writer can tune the arrangement controls, open every intermediate stage, and export only the version that still sounds like the writer (Uriostegui 2026c, 2026b).

Its most important deterministic boundary is word preservation. After a stage, the system cleans tags and annotations, compares the candidate with the original source, and reports missing words. The check cannot decide whether a performance choice has taste or cultural authenticity. It does

protect the writer from a specific and common failure: a model pass quietly dropped source words while making the output look polished.

The security boundary matters as much as the prompt design. The project is local-only because its Vite client reads a user-owned OpenAI key. A browser build containing that key must not be deployed or shared. A hosted version would need a server-side API boundary instead.

Safe-first path:

```
git clone https://github.com/hassanvfx/lyrics-refiner.git
cd lyrics-refiner
npm install
cp .env.example .env
```

Read the project README before adding a key. Use only lyrics you are authorized to share, keep .env local, and inspect every stage before exporting an arrangement.

Case study 2: Newsmusic turns a daily format into a controlled production line

Newsmusic is a creator-production system for turning news context into original, YouTube-ready music-video packages. It begins with configured channel metadata and transcripts, forms an editorial brief, develops an original song corpus and creative direction, generates music and imagery, assembles a video, and prepares delivery metadata. It is designed around a repeatable daily format: find the story, make an original interpretation, assemble the video, and make the package ready for the creator's channel. Its active orchestrator keeps third-party news-footage downloading disabled, while the default upload profile is private and review-gated (Uriostegui 2026d, 2026e).

That separation makes a powerful workflow stoppable. A creator can inspect the brief, lyrics, media, and final package before any upload. Google OAuth and generation credentials remain local, ignored by Git, and owned by the creator— not by the repository or this book. The workflow automates a chain of work; it does not automate editorial accountability.

Safe-first path:

```
git clone https://github.com/hassanvfx/newsmusic.git
cd newsmusic
python3 -m venv .venv
```

```
./.venv/bin/pip install -e '.[dev]'
git clone https://github.com/hassanvfx/kie-api-python vendor/kie-api-python
cp .env.example .env
./.venv/bin/python -m newsmusic.cli orchestrate --until video --dry-run
```

Leave the project in dry-run mode. Do not add credentials, spend generation credits, or upload anything until you understand the relevant provider terms, rights, factual-review, and channel policies.

Experiment

Use the companion guide, then perform a paper-and-terminal audit rather than a live generation:

For Lyrics Refiner, identify the original lyric, each staged artifact, and the word-preservation report. Explain one failure that the deterministic check catches and one judgment it cannot make.

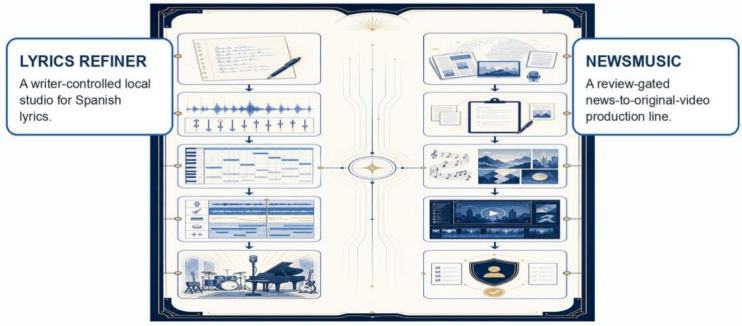
For Newsmusic, trace a single story from source context through the private review boundary. List the points where a person can stop the run.

Compare the two workflows. Mark which controls protect source integrity, credentials, external actions, factual accuracy, and rights.

Record the repository revision and documentation URL you read. Do not treat a changing external repository as a permanent benchmark result.

Two creative systems, one control contract

Generative AI becomes useful when the creator can inspect every transformation and decide what may leave the project.



Part I — work made visible

CONTEXT	Original lyric plus optional structural reference.	Configured sources, transcripts, metadata, and editorial brief.
PIPELINE	Analyze → structure → refine → annotate → arrange.	Ingest → brief → original corpus → music/image → video.
CHECK	Preservation reports flag source words dropped by a model pass.	Dry-run and configuration checks stop spend, rendering, and upload.

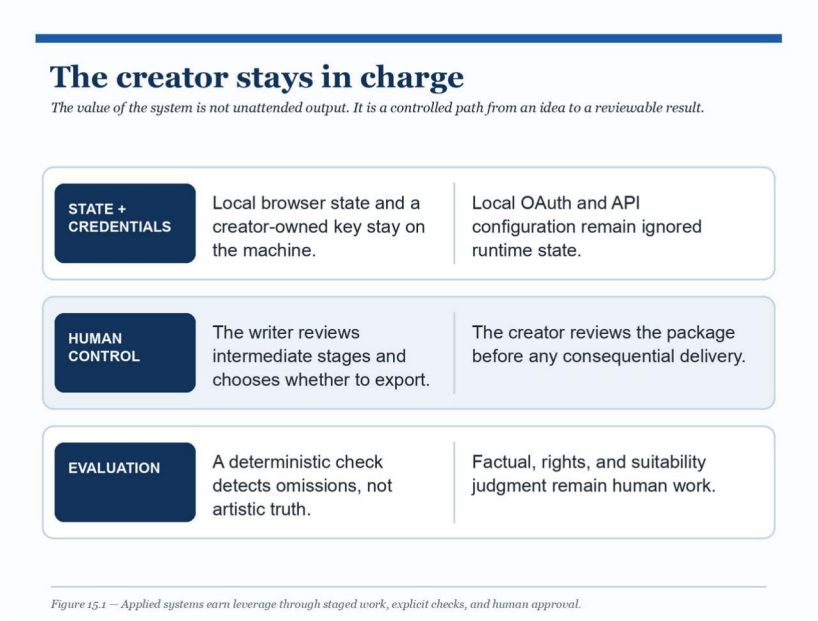


Figure 15.1 — Applied systems earn leverage through staged work, explicit checks, and human approval.

What broke

Both projects make the same general failure visible: an output can look finished while violating a constraint that was never measured. A preserved-word score can miss awkward phrasing, accidental similarity, or a poor performance choice. Dry-run success can miss factual errors, rights restrictions, poor channel fit, or an unsafe live credential configuration.

There are also project-specific traps. Lyrics Refiner must not be deployed with a VITE_* key. Newsmusic must not turn reporting context into permission to reuse third-party footage, and a private upload is still a consequential external action. In both cases, generated material needs human review before public use.

Alternatives

For lyrics, a human arranger, editor, or producer can be the right tool when the work requires cultural knowledge, performance judgment, or collaboration that a preservation check cannot provide. For creator media, a conventional editorial and production team can be preferable when rights

clearance, fact checking, and publishing scale exceed the value of automation.

At the other extreme, a one-shot prompt is useful for a throwaway sketch. It is not a substitute for a production workflow when the source text, credentials, or public distribution matter.

When to use it—and when not to

Use a staged generative workflow when you can state what must remain true, make intermediate work inspectable, and name the person who approves irreversible actions. Prefer local and dry-run paths while you are learning the system.

Do not use either case study to process material you are not authorized to share, to evade provider or platform rules, or to publish without factual, rights, and suitability review. This chapter does not promise audience growth, monetization, cultural authenticity, or legal clearance.

Evidence trail

Read the project sources and the companion articles for implementation details:

Lyrics Refiner repository and engineering story.X

Newsmusic repository and project story.X

experiments/15-generative-ai-lab/case_study_check.py and benchmarks/09-generative-ai-lab/README.md for this book's safe-first audit.

research/08-generative-ai-lab/notes.md for source boundaries and non-claims.

Takeaway

Generative AI becomes a useful production tool when it is placed inside a workflow with visible transformations, narrow checks, local credential control, and a person responsible for publication. The goal is not to remove the creator from the loop; it is to give the creator more leverage without losing the authority to say no.

Run the lab on your Mac

QR code for Chapter 15 lab

Online lab: https://hassanvfx.github.io/mac-ai/labs/15

Repository: experiments/15-generative-ai-lab/case_study_check.py

```
uv run python experiments/15-generative-ai-lab/case_study_check.py
```

Expected: Safe-first boundaries and official case-study links.

Evidence: benchmarks/09-generative-ai-lab/README.md

Scan the code to open the live lab, then use Safari Share → AirDrop to send it to your Mac. The lab follows main; clone the repository once and run the command from its root directory.

Appendix A: A Lab Notebook Protocol for Reproducible AI Work

The runnable files in this repository are not illustrations to be executed once and forgotten. They are small experiments: each one asks a bounded question, controls enough conditions to make the result interpretable, and leaves an evidence trail that another reader can inspect. This appendix is the protocol used throughout the book. Use it when adapting a chapter experiment, when adding a new comparison, or when a result seems surprising enough to deserve more than a terminal screenshot.

Start with a question that can fail

Avoid questions such as "Is MPS fast?" or "Does RAG work?" They combine too many conditions and invite a persuasive answer. A useful first question names a workload, a condition, and an observable outcome. For example: "Does the tiny network reduce its loss on the fixed synthetic dataset when the selected device is available?" Or: "Does the fixture retrieval query return the expected source path in the first four candidates?" These questions can return a useful *no*.

Write the expected result before you run the program. For a numerical exercise, that might be a tensor shape, a known gradient, or a loss that falls below a declared threshold. For a retrieval exercise, it might be a source path, citation key, or missing-evidence refusal. For a framework comparison, it might be agreement on a tiny controlled fixture rather than agreement on a headline performance number. An expected result is a debugging aid, not a license to discard failures.

State what the run cannot answer. A tiny synthetic training run can test the data/device/gradient contract; it cannot estimate production throughput or generalization. A versioned fixture can test provenance and refusal; it cannot prove semantic retrieval quality on the changing manuscript. This sentence often prevents the most common error in AI experimentation: a narrow observation quietly becoming a broad claim.

Record the experimental envelope

Every record should make it possible to reconstruct the envelope around a result. At minimum, record:

the exact command and its working directory;

the Git commit or another immutable source revision;

the seed and every deterministic setting that the program controls;

Python and relevant package versions;

operating system, hardware, and selected device;

input data identity, split, preprocessing, and model identifier where used;

the measurement boundary, warm-up policy, repeat count, and summary rule;

the expected outcome, actual result, and conditions that were not measured.

This is intentionally more than a benchmark table. A median timing without a workload is not a meaningful comparison. A model name without a revision can resolve to changed weights later. A device name without a fallback rule can hide that a CPU-only machine ran different code. The repository's benchmark records are Markdown so that the explanatory conditions live beside the values instead of being buried in a dashboard.

Use a fresh record when a material condition changes. Changing model weights, batch size, sequence length, precision, device, warm-up policy, or timing boundary creates a new experiment. Do not overwrite an earlier number so that two incompatible runs look comparable. Link both records and explain the change. Version history is useful here: it preserves the fact that an earlier observation was true under its own stated conditions even when a later run uses a better protocol.

Run in a deliberate order

Begin with the smallest deterministic route. The tensor, gradient, and Book-Intelligence fixture tests run without a downloaded model or API key, so they are useful controls on a new machine. Run the relevant test first; then run the companion program with its documented default. If the control fails, do not start installing larger optional stacks in the hope that a different backend will hide the problem.

Before an optional download, check disk space with df -h .. Before a GPU or MPS observation, print the selected device and retain the CPU fallback test. Before a provider-backed comparison, verify that the endpoint, model, and key are supplied through the documented environment variables. Never put a key in the notebook, source file, benchmark record, or shell transcript. If an optional dependency is absent, record that it was unavailable and continue with the baseline route where possible. Unavailability is a result about the environment, not a reason to invent a substitute measurement.

Make one change at a time. If a local-model run fails after increasing context, do not simultaneously change quantization, prompt, and runtime. Revert to the last known-good envelope, verify it, then vary one input. This rule is slower than speculative tweaking in the first minute and much faster once someone has to explain why two terminal outputs disagree.

Read results as evidence, not verdicts

For learning code, inspect the thing the program claims to compute. Check tensor shapes before interpreting loss. Check a gradient on a small analytic case before trusting a training curve. Check individual errors and the confusion matrix before claiming that a vision model learned a class. Check paths and excerpts before treating a RAG packet as support. The final scalar is usually the last piece of evidence, not the first.

For timing, separate setup from steady-state work. Model construction, download, compilation-like initialization, allocator behavior, and the first device call can dominate short runs. Record whether the clock includes them. Use a declared warm-up, repeat the measured portion, retain the individual values when practical, and state the summary statistic. Synchronize around an asynchronous device measurement when the runtime requires it. None of these rules creates a universal “real” number; they make the number answer a visible question.

For qualitative outputs, retain the prompt or query, the retrieved paths or model artifact, the exact response, and a reviewer judgment. Do not substitute an attractive screenshot for the whole output. Keep counterexamples and refusals in the same dataset as successes. A system that is safe only in the examples selected for a chapter has not yet earned a reliability claim.

Turn a surprise into a durable test

When a run breaks, first classify the failure. Is it an environment issue, an input/shape issue, a numerical issue, a provenance issue, an unavailable optional backend, or an unsupported claim? Preserve the smallest reproduction: command, inputs, error, environment facts, and any safe workaround. Then decide whether the failure should become a unit test, a fixture evaluation, a benchmark note, or a warning in the chapter.

A good regression test is narrow. The Book Intelligence suite, for example, asserts path attribution, citation-key preservation, blank-query refusal, empty-corpus refusal, deterministic ranking, and escaped-link reporting. It does not pretend to evaluate a language model. Likewise, the device-selection tests assert explicit MPS selection and CPU fallback without claiming that either device is faster. The test's name should state the safety or correctness property it preserves.

When a result changes after a code revision, do not ask only whether the new number is better. Ask which layer changed: input, seed, dependency, device, measurement boundary, model, chunking, or interpretation. A written record makes this comparison possible. If no layer can explain the change, label the observation unresolved and reproduce it before using it in a chapter.

A pre-commit checklist

Before committing an experiment or its manuscript claim, verify the following:

The code runs from the documented command or its limitation is stated.

The record identifies versions, seed, data/model, device, and measurement method appropriate to the claim.

The prose separates observed facts from interpretation and from open work.

A citation or committed benchmark record supports every performance, memory, framework, or model-behavior statement.

Generated caches, indexes, checkpoints, downloads, and credentials are not staged for Git.

A failure or fallback path is tested where the experiment depends on one.

The same discipline applies to editorial work. A chapter change should name the experiment or source it relies on, pass the book audit, and link to a runnable file when it describes an implementation. When a proposed AI workflow suggests a change, it stops at the human approval boundary described in Chapters 11 and 12. A plan is evidence for a review conversation; it is not permission to rewrite the book.

Closing principle

Reproducibility is not the promise that every reader gets the same number on a different machine. It is the stronger practical promise that a reader can see what was run, under which conditions, what it produced, what it did not show, and how to investigate a difference. That is enough to turn a learning project into an engineering record—and enough to keep a later agent workflow grounded in inspectable work rather than confident-looking guesses.

Appendix B: Debugging AI Experiments Without Losing the Evidence

AI experiments fail in layers. A package can be absent, a device can be unavailable, a tensor can have the wrong shape, a training run can be unstable, an index can be stale, or a generated answer can cite no evidence. The fastest way to make such failures mysterious is to change several layers at once. This appendix provides a practical order of operations for the runnable exercises in this book.

First, classify the failure

Start by writing one sentence that says what failed and at which boundary. "The experiment did not work" is not a classification. "The optional TensorFlow group is not installed," "the requested MPS device is unavailable," "the model output shape differs from the target," and "the query returned no grounded evidence" are classifications. Each points to a different next check.

Use five broad categories:

Environment: an executable, dependency group, model artifact, storage allocation, device, or API configuration is unavailable.

Data contract: inputs, labels, shapes, dtypes, normalization, split, or corpus paths are not what the code expects.

Numerical behavior: a gradient is wrong, loss diverges, a result is nondeterministic, or a timing boundary is not valid.

Provenance: an evidence path is missing, a citation key is lost, an index is stale, or a Markdown link leaves the configured corpus.

Policy: a model output lacks evidence, attempts to widen its authority, or a proposed action reaches an approval boundary.

Classification is not bureaucracy. It limits the next action. An environment failure calls for an installation or fallback check; it does not call for a new architecture. A provenance failure calls for rebuilding or reviewing the

index; it does not call for a more fluent model response. A policy failure is a successful stop condition, not something to optimize away.

Reproduce the smallest case

Run the narrowest test or command that exercises the failed contract. For a tensor broadcasting issue, begin with experiments/01-tensors/broadcasting.py or its matching test, not a full neural-network training loop. For gradient confusion, use the scalar autograd example before inspecting an optimizer. For retrieval, use the frozen fixture tests before indexing the live manuscript. For approval, use the no-write checkpoint test before considering a writer.

Reduce inputs while retaining the error. A single batch, one known text query, or one fixture document is easier to inspect than an entire dataset. Preserve the exact seed, command, error text, and environment facts. If reducing the case changes the behavior, record that fact too; it tells you that scale, ordering, or initialization may matter.

Do not make an optional dependency a prerequisite for basic diagnosis. The repository deliberately retains deterministic CPU-capable controls. If an embedding model cannot download or a local runtime is absent, the fixture retrieval path can still test chunking, path attribution, citation propagation, and refusal. If MPS is unavailable, device-selection tests can still confirm that the program labels the fallback rather than silently changing the request.

Inspect contracts in order

For a learning pipeline, inspect inputs before outputs. Print or assert shape, dtype, device, and a small value sample. Then inspect the model output shape, loss input order, gradient presence, and parameter update. Only after those contracts are true should you reason about a training curve. A falling loss on misaligned labels is not evidence that a model learned the intended task.

For a benchmark, inspect the workload before the number. Identify model, input dimensions, batch size, precision, selected device, warm-up policy, repetitions, and synchronization. A number that cannot be tied to these facts is an observation to repeat, not a number to put in a comparison table. Keep cold start, model download, and steady-state work separate whenever the question distinguishes them.

For Book Intelligence, inspect corpus membership before relevance. A returned source must be inside the declared repository subtrees and resolve to a current file. Then inspect the excerpt and its citation keys. A high-ranked chunk is a candidate, not support. If there is no positive evidence, the correct answer is the missing-evidence message. Never repair a retrieval miss by letting a model invent a path or an answer.

For a structured plan, inspect the trusted allow-list before its prose. The validator may accept only retrieved paths, overwrites caller-controlled fields, and always restores the human-approval requirement. A syntactically valid JSON object that names outside.md is an unsafe output that has been correctly rejected, not a mostly successful plan.

Make installation failures explicit

Before installing an optional group, check both free disk space and the scope of the dependency. The environment setup chapter documents the commands; the important debugging principle is to state what is missing and what remains available. For example: "The TensorFlow group is not installed; the PyTorch and deterministic tests remain runnable." This keeps a learning session moving without pretending that a comparison was executed.

If installation is appropriate, use the project-managed command and rerun the narrow test first. Do not mix an installation, a package upgrade, a source change, and a new model download in one unrecorded step. If storage is insufficient, stop before the download and record the estimate or available space. Freeing storage is a user decision when it could affect unrelated data.

Credentials deserve a stricter rule. A missing API key or endpoint should fail before a network call and should never appear in an error report. Record only non-secret facts needed for reproduction: adapter type, model identifier, endpoint class, package version, and the fact that credentials were absent or invalid. The deterministic test suite is the correct fallback until a controlled provider experiment is deliberately configured.

Turn the repair into evidence

When you identify a fix, preserve the before-and-after conditions. Add a unit test when the failure expresses a stable correctness or safety property. Add a fixture case when it needs frozen source-shaped input. Add a benchmark record when the observation depends on a machine, workload, or timing

method. Add a chapter warning when the reader is likely to encounter the same confusion.

Keep the repair scoped. A test that proves blank queries refuse an answer should not also assert model quality. A test that proves MPS falls back to CPU should not claim a performance advantage. A review test that catches an escaped link should not read the target outside the corpus. Small tests make a later failure diagnosable and prevent an attractive demo from becoming a broad, untested guarantee.

Finally, update the active project journal before committing. Record what changed, commands run, result, limitations, and the next action. The journal should link to detailed benchmark or research records instead of pasting raw terminal output. That leaves the next work session with a decision trail and keeps the canonical manuscript focused on what readers need to learn.

A compact triage loop

State the failed contract in one sentence.

Reproduce it with the smallest fixture or test.

Inspect environment, inputs, and provenance before changing a model.

Change one declared condition.

Re-run the narrow check, then the relevant suite.

Record the result, limitation, and whether it became a test, fixture, or benchmark.

The point is not to eliminate failure. It is to make failure teach us something that a later reader can reproduce and a later agent cannot quietly obscure.

Appendix C: Evaluation and Release Gates

A beta is a defined set of artifacts and checks at a named revision. It is not a promise that every future capability is measured. This appendix separates four kinds of evidence: unit tests establish small contracts; fixture evaluations establish reproducible provenance and refusal behavior; benchmark records establish observations on a declared machine and workload; human review establishes whether evidence supports a claim or a print page is acceptable. None can substitute for the others.

Code and evidence gate

Run the baseline checks from a synchronized environment:

```
uv sync --group dev
uv run ruff check .
uv run pytest
uv run python evals/run_reliability.py
make audit-book
```

The test suite covers numerical expectations, CPU fallback, framework and transformer helpers, retrieval contracts, structured-output validation, approval state, and workflow comparisons. The no-secret reliability runner uses frozen fixtures to check source paths, citation keys, missing-evidence refusal, deterministic ranking, and unsafe-link reporting. The book audit checks canonical chapter sections, citations, links, runnable paths, research references, and the declared word budget.

Read each failed gate as a routing signal. A broken code path means prose and repository drifted. A missing citation means the claim needs evidence or a narrower wording. A fixture failure means a provenance or policy invariant changed. A word-budget failure means the manuscript is not at its planned beta scope, even if every unit test is green. Do not replace a failed gate with a manual claim that the system “mostly works.”

Site gate

The Docusaurus site presents canonical Markdown; it is not a second manuscript. Build it after source or navigation changes:

```
cd site
npm ci
npm run build
```

A successful build proves that the current site can be generated. It does not publish anything. Publishing requires a chosen GitHub destination, deliberate Pages configuration, a remote, and an intentional push. Those are release decisions; neither an exercise nor an agent workflow should make them by default. Inspect generated reading flow, diagrams, and code links before a human release decision.

Manuscript gate

Build the canonical Markdown with the versioned Lulu reference template:

```
./scripts/build-book.sh
```

This creates an ignored DOCX. It proves that citations, assets, and the template can produce a Word document from the current source. It does not prove that the PDF is printable. An interim DOCX render is valuable editorial evidence, but the release path still requires a Microsoft Word PDF export on macOS, PDF preflight, and visual inspection of every page at print scale.

Preflight the actual upload PDF for the 6×9 target, single-page layout, embedded fonts, and detectable image-resolution concerns. Treat every issue as a source/layout correction followed by another export, not a cosmetic exception to the evidence trail. Generated DOCX/PDF artifacts remain untracked during editing.

Final page count is a dependency: download the exact paperback cover template only after it is frozen, because spine width depends on it. Cover artwork, metadata, ISBN, upload, and proof ordering remain human choices. A local build must never be described as a Lulu submission or approved proof.

Optional provider-comparison gate

API-backed comparisons begin only when a provider, model, endpoint, and evaluation protocol are chosen. Credentials are environment configuration, never fixture, source, journal, or benchmark content. Fix the task set, corpus revision, retrieved evidence, prompt/schema version, model/settings, retries, and timing boundary. Record successes, refusals, malformed outputs, timeouts, and configuration failures; score citation accuracy, plan completeness, review coverage, latency, cost, and unsafe-action refusal under that same protocol.

The no-write rule survives every provider choice. A model may propose only from supplied evidence. Validation rejects unapproved paths and restores the human approval requirement. Any source, Git, journal, or external action needs a separate, scoped approval.

Release checklist

The manuscript meets scope and passes its editorial/evidence audit.

Python tests and deterministic evaluations have passing records.

The site builds from canonical Markdown.

DOCX builds through the committed template and receives layout inspection.

The final Word PDF is preflighted and every page is reviewed.

Release notes distinguish observations from unmeasured work.

After beta, freeze the text, copyedit, finalize page count, create the cover from Lulu's exact template using the assigned ISBN metadata, and order a Lulu proof. Proof findings create new tracked corrections and another preflight. A successful build is necessary; an approved proof is the evidence needed before publication.

Appendix D: Guided Exercises and Evidence Prompts

These exercises are deliberately small. Their purpose is to make a reader observe a contract, record the conditions, and explain a failure before moving to the next abstraction. Each exercise has a runnable companion file, but the terminal output alone is not the answer. Record the command, source revision, device, expected result, actual result, and one limitation in your lab notebook.

1. Broadcasting is an alignment rule

Run experiments/01-tensors/broadcasting.py. Before running it, write down the shapes you expect for every operand and the resulting tensor. Then change one axis in a copy of the exercise so that two non-singleton dimensions no longer agree. Predict whether the failure is caused by values, dtype, device, or shape; run it; and retain the error message.

The evidence to collect is a shape table, not a vague statement that "broadcasting happened." Explain which axis was implicitly repeated and why the invalid case could not be repeated. Do not turn this into a performance test; it is a data-contract exercise. Restore the original source after the experiment or work in a disposable copy.

2. Derivatives are local evidence

Run experiments/02-gradients/autograd.py. Identify the scalar objective, input value, and expected derivative before inspecting the output. Change the input in a copy and calculate the new derivative by hand. If the automatic and manual values disagree, inspect the definition of the objective and the point at which the derivative is evaluated before adding an optimizer or a neural network.

Write one sentence separating the claim the run supports from the claim it does not. A correct scalar gradient supports that this small autograd path behaves as expected. It does not establish that a larger network will train, that an optimizer is configured correctly, or that a chosen loss is appropriate for a dataset.

3. A training curve needs a control

Run experiments/03-pytorch/train_tiny_network.py. Record the printed seed, selected device, initial/final loss, and any declared accuracy or prediction check. Run it a second time without changing conditions. If the output differs, identify whether the script controls random seeds, device selection, or data ordering before interpreting the difference.

Next, make one controlled change: reduce epochs, alter learning rate, or change the synthetic data size. Do not change more than one. Record the new envelope in a separate observation rather than overwriting the first. Explain whether the new result answers the same question. A shorter run may be useful for a smoke test yet incomparable to the original training observation.

4. Device selection is not a speed claim

Run the Day 1 tests and the training example on the available machine. Note whether MPS or CPU is selected and whether the requested-device fallback is explicit. On a machine without MPS, the useful result is a clearly labelled CPU route, not a failure to be hidden. On a machine with MPS, a successful run is still not a PyTorch-versus-CPU benchmark.

If you measure timing, define the workload first: model, data shape, batch size, precision, warm-up policy, repeat count, synchronization rule, and clock boundary. Keep construction and download out of a steady-state measurement unless cold-start experience is the declared question. Store individual timing values and name the summary. If you cannot state these conditions, record no performance conclusion.

5. Compare frameworks by invariant

Run the PyTorch and TensorFlow vision fixture commands documented in Chapters 4 and 5. Confirm that data split, seed, labels, metric definition, and expected fixture output are matched before comparing any result. The first useful check is whether both implementations satisfy the small shared task. A difference means inspect preprocessing, label mapping, output activation/loss convention, and update loop before attributing it to the framework.

Write a two-column note: “observed under this fixture” and “not established.” For example, agreement on a synthetic classification contract may be observed; general vision quality, model ecosystem, memory use, and throughput are not. This habit protects a comparison chapter from becoming an API preference essay.

6. Trace a tokenizer to a label

Run experiments/06-transformers/inspect_sentiment.py only after checking available disk space and recording the model revision/configuration. Inspect the input text, token IDs, special tokens, attention mask, raw logits, label map, and ranked output. Compare the manual route with any high-level pipeline only as an interface check: both paths should use the same model and label map.

When the labels disagree, work backwards from the label map to the logits and then to tokenization. Do not begin by declaring a model hallucination. Record the exact input and model identity, and state whether model weights were newly downloaded. A single classification is an inference trace; it is not a calibration or safety evaluation.

7. Treat retrieval results as candidates

Run the deterministic route:

```
uv run python experiments/08-embeddings/book_search.py --deterministic \
  --query 'Where are benchmark limitations recorded?'
```

Inspect every returned path. Does it exist inside the configured corpus? Does the excerpt answer the question, or merely share a word? Identify one relevant candidate and one possibly misleading neighbor. Then issue a blank query or use the frozen evaluation suite to observe the refusal path. Record why missing evidence is safer than a guessed answer.

If you install the optional learned embedding group, pin the package/model, check storage first, and keep the deterministic result as a control. Compare paths and relevance judgments, not raw scores across unlike encoders. A score is a ranking signal within an encoder/corpus combination; it is not a percent probability of truth.

8. Audit a grounded evidence packet

Run experiments/09-rag/grounded_answer.py --deterministic with a question that has a known source. For each displayed excerpt, check path resolution, citation-key proximity, and whether the excerpt supports the sentence you want to write. Then ask an unsupported question. Preserve the missing-evidence message; do not add a web answer or a fluent summary to make the demo look more useful.

Write a proposed one-sentence answer only after attaching its source path. If the excerpt does not entail the sentence, rewrite it as a narrower statement or leave it unanswered. This is the human judgment that structural tests cannot perform for you.

9. Reject a well-formed unsafe plan

Run experiments/10-systems/compare_structured_planning.py with its fixture. Inspect the evidence allow-list, rejected path warning, caller-controlled objective, and approval flag. Explain why a JSON object containing an unretrieved path is unsafe even if it parses successfully. Then inspect the empty-evidence case in the tests: steps should be cleared instead of converted into generic write-like recommendations.

Your exercise output is a short contract table: input evidence, accepted paths, rejected paths, warnings, and approval requirement. Do not configure an API key for this exercise. The deterministic route is sufficient to learn the safety boundary, and it keeps the result reproducible without a provider decision.

10. Resume without executing

Run the no-write LangGraph example and its tests. Pause the graph, inspect the approval payload, then resume once with rejection and once with approval. In both cases, confirm the terminal status explicitly says no write and that the source fixture has not changed. Explain why approval of a proposal is not permission to apply a diff.

Finally, choose one exercise whose result surprised you. Turn it into a small record with command, revision, expected result, actual result, limitation, and next action. That record—not the fact that the command eventually passed—is the completion artifact for this appendix.

Appendix E: Working Glossary

Approval boundary. A point at which a workflow stops and requires a human decision before any consequential action. In this project, approval can resume a no-write proposal state; it is not standing permission to edit files, commit, or use an external service.

Attention. A mechanism that combines token representations according to content-dependent weights. In a transformer, queries, keys, and values make the relationship between tokens explicit for the current input.

Autograd. Automatic differentiation: a system records differentiable operations and computes derivatives of a scalar objective with respect to selected inputs or parameters.

Batch. A group of examples processed together. Batch size changes the shape, memory demand, gradient estimate, and often the timing question; it is therefore part of an experiment envelope.

Benchmark record. A committed description of an observed run that names workload, machine/device, versions, timing method, result, and limitations. It is evidence for its own conditions, not a general product ranking.

Checkpoint. Persisted operational workflow state used to resume a paused process. It is not the canonical manuscript, not a Git history, and not proof that the evidence referenced by an old proposal remains current.

Chunk. A bounded unit of indexed source text. This project uses paragraph groups with a visible character limit so returned excerpts retain enough local context to be inspected.

Citation key. The identifier inside a citation marker or BibTeX entry. A retrieval result may preserve a key found in its excerpt; that preservation does not mean the key supports every sentence someone might write about the excerpt.

Context window. The amount of tokenized input and output a model can handle for one request. It is a model/runtime constraint, not a guarantee that all provided context is relevant or used correctly.

Corpus. The configured set of documents eligible for indexing or retrieval. The Book Intelligence Assistant uses allowlisted repository subtrees rather than following arbitrary filesystem paths or Markdown links.

Cosine similarity. For normalized vectors, the dot product measuring their directional alignment. It ranks candidates inside a chosen encoder and corpus; it is not a calibrated probability that a source is true or answers a question.

Deterministic fallback. A documented route whose behavior does not depend on an optional model or provider. It keeps core tests and safety checks useful when a dependency, credential, model download, or accelerator is unavailable.

Device fallback. Explicitly selecting a safe supported device when a requested accelerator cannot be used. It must be labelled so a user does not mistake a CPU run for an MPS observation.

Embedding. A numeric vector representing an item such as text. A learned embedding may capture useful relationships, while the repository's hash-vector baseline exists to make contracts repeatable without claiming semantic quality.

Entailment. Whether supplied evidence actually supports a proposed claim. Path resolution, citation-key propagation, and schema validation are useful structural checks, but a reader still judges entailment.

Evidence packet. The retrieved paths, excerpts, metadata, and citation keys shown before or instead of a generated answer. A packet is inspectable context, not a conclusion.

Fixture. A small, frozen test input with known expected properties. Fixtures make source attribution, failure behavior, and workflow contracts testable even while the live manuscript changes.

Gradient. The derivative of an objective with respect to parameters or inputs. Gradient descent uses it to select a local update direction; a correct gradient alone does not prove a model will generalize.

Grounding. Constraining an answer or proposal to retrieved, inspectable evidence. Grounding is stronger than merely displaying a citation, but it still requires evaluation of whether a cited excerpt supports the claim.

Human in the loop. A system design in which a human makes a named decision at a persisted, reviewable transition. It is not a decorative confirmation button placed after an autonomous write already occurred.

Index. A data structure derived from a corpus to support search. An index is a cache of a particular source revision and becomes stale when its inputs change; it is not the authority over current source files.

Inference. Computing model outputs from inputs using fixed parameters. It differs from training, which also computes gradients and updates parameters.

LangGraph. A framework for defining stateful graph workflows. In this book it illustrates explicit nodes, edges, checkpoints, interrupts, and no-write approval states; it is not itself a permission system.

Loss. A scalar objective that quantifies disagreement between model outputs and targets under a declared rule. Its absolute scale depends on the chosen loss and data, so values from unlike runs are not automatically comparable.

MPS. Apple's Metal Performance Shaders backend used by PyTorch on supported Apple Silicon systems. Its availability and measurement boundaries must be recorded before interpreting a local observation.

Normalization. Transforming values to a declared scale. L2-normalizing a nonempty vector makes its dot product with another normalized vector equal to cosine similarity; normalizing input data changes a model's effective problem.

Prompt injection. Untrusted text attempting to change an assistant's instructions or authority. Retrieved repository material is evidence to cite, not authority to override corpus boundaries, no-write rules, or approval.

Provenance. Information that lets a reader trace a result to its source, such as repository path, chapter identifier, citation key, experiment metadata, corpus revision, or benchmark conditions.

RAG. Retrieval-augmented generation: retrieving context at answer time and using it to constrain a response. It does not retrain the model or make the response automatically truthful.

Recall at k. For a question with acceptable sources, whether at least one acceptable source appears among the first *k* retrieved candidates. It measures a retrieval condition, not whether a generated answer is fully supported.

Reference template. A Word document that supplies the intended print layout to a DOCX conversion. The Lulu template governs the generated manuscript's layout; final PDF inspection is still required.

Regression test. A test retained because a specific failure or invariant must not silently return. A narrow regression test is more diagnosable than a broad demonstration that blends many possible causes.

Retriever. A component that ranks or selects corpus candidates for a query. It should preserve the original evidence record rather than returning only detached text or a score.

Seed. A value controlling pseudo-random choices. It improves repeatability when recorded with versions and inputs, but cannot remove every hardware or runtime source of nondeterminism.

Structured output. Model output parsed into named fields under a schema. The schema checks shape; application validation must still enforce evidence allow-lists, caller-controlled objectives, and human approval.

Token. A unit produced by a model's tokenizer. Tokens are not necessarily words; special tokens, padding, and masks are part of an inference trace.

Trace. A record of workflow inputs, state transitions, outputs, timing boundaries, and errors. A safe trace omits credentials and does not turn raw private context into a permanent log by default.

Vector store. A persistent system for searching vector embeddings. It may be useful for a larger or multi-user corpus, but it adds operational cost and does not remove the need for source provenance and evaluation.

Warm-up. Unmeasured work performed before a timing sample to separate setup effects from a declared steady-state measurement. It must be stated, not silently applied, because it changes the question a benchmark answers.

Appendix F: Book Intelligence Capstone Walkthrough

The Book Intelligence Assistant is the book's canonical capstone because its corpus, failure cases, and limits are all visible in the same repository that teaches the techniques. This walkthrough traces one maintenance task through retrieval, evidence-only answering, planning, critique, approval, and evaluation. At no point does the workflow modify prose, code, Git state, or an external service.

The task

Suppose an editor asks: "Review the embeddings chapter and propose the smallest evidence-backed improvement." This is not an instruction to rewrite a chapter. It is a request for a reviewable proposal. The task enters the system with a clear objective and a declared corpus boundary: research notes, canonical chapters, experiments, and benchmark records. Private files, arbitrary filesystem paths, and the Internet are outside that boundary.

The first question is whether the corpus contains useful evidence. The retriever chunks allowed files, attaches repository path, corpus kind, chapter identifier, citation keys, and lightweight metadata, then returns a small ranked candidate set. A similarity score only routes attention. The editor opens the returned paths and excerpts before deciding whether they support the maintenance task.

Evidence before prose

The grounded-answer layer renders an evidence packet: each excerpt appears under its repository path, with citation keys where the same chunk contains them. If retrieval is empty, blank, or too weak for the declared policy, it returns the fixed missing-evidence response. It does not turn an absent source into a generic answer, a guessed citation, or a provider call.

This gives the editor a useful checkpoint. A path that merely shares words with the question is not automatically relevant. A real citation key is not automatically support for every conclusion near it. The editor can narrow

the question, add missing research, inspect an exact identifier, or stop. The system has not yet earned a recommendation.

Plan and critique

When evidence exists, the planning layer receives only the objective and the retrieved allow-list. Its structured proposal names evidence paths, steps, and unsupported-claim warnings. Validation removes any path not retrieved, restores the caller's objective, and forces approval_required to true. With no evidence it clears action-like steps. A well-formed object is therefore not enough to become an authorized plan.

The critic then examines the same scope plus deterministic editorial findings. It can flag missing alternatives, unresolved or unsafe links, missing experiments, and unsupported claims when those findings are present in the evidence. It cannot silently widen the corpus or write a fix. A direct SDK adapter, a LangChain adapter, or a local model may produce wording differently, but each must pass through this same evidence and approval contract.

Persisted review, not execution

The LangGraph example persists a proposal under a new thread ID and interrupts with an approval payload. The payload names objective, evidence paths, plan, critique, and the explicit statement that no source, Git, or external action will occur. A rejection ends rejected_no_write; an approval ends approved_no_write. Both outcomes are durable and neither edits a file.

This deliberately modest endpoint teaches the critical separation: a reviewer can accept the *idea* of an evidence-backed revision without granting a broad writer capability. A future writer would need fresh evidence checks, a diff-scoped second approval, and access limited to one approved target. The beta implements none of those write operations.

Evaluation as a capstone check

The frozen fixture corpus supplies deterministic cases for source attribution, citation-key preservation, grounded paths, unsupported questions, blank input, empty corpora, deterministic rank order, unsafe links, planning no-write behavior, and critic findings. The workflow-comparison fixture runs a

deterministic control, a single planner, and a researcher/critic/writer shape against the same task. Every route must preserve the expected evidence path, flag the known omission, require approval, and report no write.

Run the capstone's baseline evidence gate with:

```
uv run pytest tests/test_book_intelligence.py \
  tests/test_book_intelligence_evaluation.py \
  tests/test_approval_workflow.py tests/test_workflow_comparison.py
uv run python evals/run_reliability.py
```

Passing this gate does not claim a general agent-quality result. It establishes that the repository's canonical example behaves safely under its versioned fixture contract. A model-backed comparison remains an optional future experiment with a selected provider, declared settings, redacted traces, and separate latency/cost/quality evaluation.

What the reader should retain

The capstone is not a chatbot that knows the book. It is a chain of explicit boundaries: corpus membership, provenance-preserving retrieval, evidence-only grounding, structured validation, critique, persisted human decision, and no-write termination. Its usefulness comes from making each boundary testable. That same shape transfers to a real engineering project only after its corpus, permissions, retention policy, and evaluation dataset have been chosen with the care they deserve.

Appendix G: Comparison Methods Without False Rankings

The word *compare* is deceptively simple. Two programs can both produce a number while answering different questions; two model outputs can look similar while using different data; two agent workflows can both sound helpful while one has silently widened its authority. This appendix is a method for making comparisons useful rather than theatrical.

Begin with the decision

Write the decision the comparison should inform. “Choose a framework for a small teaching classifier,” “decide whether learned retrieval improves a fixture task,” and “decide whether a critic role catches a known omission” are decisions. “Find the fastest model” is not yet a decision, because speed depends on model artifact, workload, device, setup policy, precision, batch, sequence length, and what the user values.

Once the decision is named, list the outcomes that matter. A framework exercise may care first about matched predictions and failure inspection. A local-model experiment may care about whether the model fits, follows a fixed prompt contract, and remains responsive under a declared workload. A retrieval change may care about acceptable-path recall and citation preservation. An agent workflow may care about review coverage and unsafe-action refusal. Do not add a metric because a tool happens to print it.

The comparison should be able to recommend “no difference established” or “keep the simpler control.” That result is especially valuable in a learning repository. It prevents an unmeasured preference from entering a chapter as a technical conclusion.

Hold the treatment still

For a framework comparison, hold data identity, train/validation/test split, label mapping, preprocessing, seed policy, model capacity, objective, optimizer semantics where comparable, metric definition, epoch/batch policy, and reporting boundary as still as practical. Some APIs will differ. Record those differences instead of pretending the programs are identical.

The first question is often narrower: do the implementations satisfy the same small fixture contract?

For a device comparison, hold code path, model, input shapes, batch size, precision, warm-up policy, repeat count, synchronization, and timing boundary. Record OS, package versions, hardware, memory conditions if observed, and whether the model was already cached. Separate model download and construction from steady-state work unless cold start is the explicit treatment. A result from a different workload is a new record, not the other row of the same table.

For retrieval, hold corpus snapshot, file allow-list, chunking policy, query set, acceptable-path judgments, ranking limit, and scoring rule. A learned encoder and a deterministic baseline may legitimately use different vector representations; compare their returned paths and human relevance judgments, not raw score magnitudes. Cosine values from different encoders are not a common probability scale.

For workflow shapes, hold objective, corpus revision, retrieved evidence, editorial checklist, output schema, approval rule, and evaluator fixed. If a researcher/critic/writer graph receives richer context than a single planner, then a changed outcome is not evidence that role specialization alone caused the change. Compare a deterministic control first, then introduce one distinct handoff whose failure-prevention value can be tested.

Define success and failure before looking

Pre-register the smallest scorecard useful for the decision. For a classifier fixture, include expected prediction/metric agreement and a selected error case. For retrieval, include acceptable path at k, source resolution, citation-key preservation, and a known unsupported query. For structured planning, include allowed-path adherence, parse-failure visibility, missing-evidence behavior, and approval requirement. For workflows, include path attribution, review coverage, no-write behavior, and explicit terminal status.

Define failures too. A timeout, missing model, malformed response, unsupported path, empty retrieval, device fallback, or out-of-memory result is part of the comparison data. Do not remove it from a summary because it makes one row less attractive. If a condition cannot run, report it as unavailable under the declared envelope. Another system running a different workload does not repair that missing observation.

Use separate columns for observation and interpretation. “Both fixture implementations classified the controlled examples correctly” is an observation. “Framework A is better for vision” is a broad interpretation that the fixture does not support. “The learned encoder placed an acceptable source in the top four for this frozen question set” is an observation. “It understands the book” is not.

Measure with visible boundaries

Timing begins and ends somewhere. State where. If device work is asynchronous, synchronize according to the runtime before measuring the wall-clock interval. State whether warm-up was performed and whether a value is a minimum, median, mean, percentile, or full list of repeats. State whether retries occurred. A median over five warmed runs can answer a modest steady-state question; it does not reveal first-use experience, tails, energy, multi-user contention, or memory headroom.

Quality also needs a boundary. For a small RAG evaluation, create a versioned question set before tuning, name acceptable and insufficient source paths, and retain reviewer judgments. For a plan comparison, define the required sections and evidence links. For a qualitative inference comparison, retain exact input, model revision, decoding settings, output, and a rubric. A screenshot or one attractive completion is evidence for a demonstration, not a quality rate.

Cost must be measured under the same protocol as quality and latency. API requests may add retries, input tokens, output tokens, and provider-specific billing; local runs may add model-download, disk-cache, memory, and thermal conditions. Record what was actually observed. Do not infer cost from a model’s parameter label or infer memory use from a marketing figure.

Read a comparison table responsibly

A table should make mismatch visible. Put the workload, source revision, device/model, measurement boundary, and limitation near the result. Use “not measured” instead of leaving a reader to infer a zero or a win. Keep different evidence types separate: a unit-test pass is not a millisecond result; a reviewer judgment is not a package feature; an unavailable dependency is not a failure of the underlying technique.

Prefer paired observations. Run two alternatives on the same case, preserve the same trace fields, and inspect differences one row at a time. If an unexpected result appears, return to the envelope: input, seed, version, model, device, warm-up, corpus, prompt, schema, or evaluator. Do not immediately add more abstractions or rerun until a preferred result appears.

If the comparison cannot support a decision, say so. The correct next step may be a smaller fixture, a more precise workload, additional evidence cases, or no comparison at all. This is not a weakness. It is the point at which an honest engineering record remains more useful than a false ranking.

A reusable comparison record

For every row, retain:

Decision and question being answered.

Alternative, exact version/model, and configuration.

Input/corpus/data snapshot and expected outcome.

Device/environment and dependency versions.

Measurement or evaluation method, including warm-up/retries.

Raw observations or a link to their committed record.

Result, limitation, and the next decision it supports.

Commit the record when it supports manuscript prose. Keep generated caches, local indexes, credentials, and bulky artifacts out of Git. A reader should be able to reproduce the *question* and investigate a difference, even if their machine cannot reproduce the same absolute value.

Closing principle

The best comparison is not the one with the most rows or the strongest headline. It is the one whose alternatives answer the same declared question, whose failure cases remain visible, and whose conclusion stops exactly where the evidence stops. That discipline is what allows this book to compare tensors to models, models to workflows, and workflows to human control without turning any of them into mythology.

Appendix H: Ten Days of Learning by Building

This is a practical sequence for turning the book into a working learning project. Each day has a narrow technical focus, a visible repository output, and a gate. The gate is not a grade or a deadline: it says what evidence should exist before the next abstraction is allowed to hide the previous one.

Day 0 — Make the baseline visible

Confirm Python, uv, Node, Pandoc, and Word. Run the existing Python tests and site build before changing source. Read the active project journal, record the source revision, and check disk space before any optional download. The output is a short baseline record that says what is installed and what remains available when an optional group is absent. A small deterministic core is more useful than a collection of untested packages.

Day 1 — Tensors, derivatives, and one controlled model

Run the broadcasting and autograd examples, then the tiny PyTorch training program. Predict tensor shapes and scalar gradients before looking at output. Record seed, package version, selected device, input, expected result, actual result, and limitation. The gate is correct numerical behavior and an explicit CPU fallback. A loss curve or an MPS label does not yet establish speed or generalization.

Day 2 — Learn to inspect a vision error

Run the small CNN fixture and preserve data split, preprocessing, label map, metric, and confusion matrix. Inspect one wrong prediction. The output is a reproducible error-analysis record and a chapter explanation of what that error does and does not show. The gate is an interpretable pipeline contract, not a claim that a tiny fixture makes the architecture competitive.

Day 3 — Compare framework contracts

Install the optional TensorFlow group only after checking space. Match data, split, seed policy, labels, model intent, and metric definition with the PyTorch fixture. Investigate differences from preprocessing through loss semantics before attributing them to a framework. The gate is a limited shared fixture observation. If a runtime is unavailable, record that fact and retain the control rather than inventing an incomparable table.

Day 4 — Trace a transformer input to its label

Run tokenizer inspection and a declared inference route. Inspect text, token IDs, special tokens, mask, logits, label map, and ranked output. Compare a manual path with a convenience pipeline only as an interface check. The gate is an inspectable token-to-label trace. One output does not establish calibration, safety, or broad language understanding.

Day 5 — Treat Apple Silicon as an experiment envelope

If a local model runtime and public model are available, record artifact, quantization, prompt, output limit, cache condition, warm-up, and timing boundary. Keep download and construction separate from steady-state timing unless cold start is the question. The gate is an honest local observation, including an unavailable-model or out-of-space result. It is not a hardware ranking or a claim about longer contexts.

Day 6 — Retrieve evidence before generating prose

Begin with the frozen deterministic fixture. Verify expected paths, citation keys, blank-query refusal, empty-corpus refusal, deterministic ranking, and unsafe-link reporting. Only then index the live corpus locally. The gate is provenance: every result resolves inside the declared corpus and every answer shows evidence or refuses. Learned embeddings inherit this contract and need a separate relevance set before any quality claim.

Day 7 — Validate plans as data, not authority

Exercise direct-shaped and LangChain structured paths over the same retrieved evidence. Inspect rejected paths, restored caller objective, missing-evidence behavior, and the forced human-approval flag. The gate is that a valid JSON object cannot widen authority. Keep provider configuration out of the exercise until a model, endpoint, redaction policy, and evaluation protocol are deliberately selected.

Day 8 — Resume without writing

Pause the persisted workflow, reopen its checkpoint, and resume one thread with rejection and another with approval. Inspect the payload and terminal states. Verify that the fixture source is unchanged in both cases. The gate is durable, reviewable no-write state. A future writer would require fresh evidence, a constrained diff, and another scoped approval.

Day 9 — Ask whether extra roles earn their cost

Run deterministic, single-planner, and researcher/critic/writer shapes against the same fixture. Compare evidence paths, editorial finding, approval, and no-write status. The gate is a visible contract across all shapes. Retain the smallest workflow unless a versioned evaluation demonstrates that a distinct handoff catches a useful failure often enough to justify its coordination cost.

Day 10 — Produce a reviewable beta candidate

Run Ruff, the full tests, deterministic reliability suite, book audit, and site build. Build the DOCX from canonical Markdown and the committed Lulu template; render it for editorial inspection. Update the journal with commands, results, limitations, and next action. The output is a beta candidate, not automatic publication.

GitHub publishing requires a chosen remote and intentional deployment. Print release requires a Word-exported PDF, full preflight and page inspection, frozen page count, a final Lulu cover template carrying the assigned ISBN, and a Lulu proof. These remain human-owned decisions.

Daily habit

Start with the journal, run the narrowest reproducible control, and change one declared condition at a time. Before committing, capture changed files, commands, observations, failures, and the next step. Link detailed benchmark and research records instead of copying raw terminal output into prose. The transferable skill is not memorizing a framework: it is leaving an evidence trail that another reader can inspect and improve.

After the first sprint: turn activity into a maintained project

Finishing a sequence of exercises creates many small artifacts: commands that worked, commands that did not, scratch observations, cached models, questions that led nowhere, and a few results worth carrying into the manuscript. The next job is editorial selection. A repository becomes a useful companion when a reader can tell which artifact is a stable lesson, which is a measured record, which is a generated cache, and which is merely a future idea.

Start with the canonical chapter, not the terminal history. For each chapter, ask four questions. What concept does the reader need before running anything? What smallest program demonstrates the mechanism? What real implementation or fixture makes the mechanism useful? What failure, alternative, and usage limit prevent the lesson from becoming a slogan? If a section cannot answer one of these questions, it may belong in a research note or journal instead of the published prose.

Then link every substantive claim to an evidence path. A training paragraph should link to the experiment and the observation record that supports its scope. A framework comparison should link to matched fixtures and state the conditions that were held fixed. A retrieval claim should link to a fixture case, source paths, and the rule that governs missing evidence. An agent claim should link to its no-write workflow test. Links are not decorative references: they are invitations for a reader to inspect the exact boundary of the claim.

Keep experimental records factual. A good record says that a named model, version, device, prompt, workload, and timing method produced an observation on a certain date. It also says what was not measured. Avoid

retrospective prose that turns an early run into a universal finding because later chapters happen to need an example. If a result needs a stronger claim, design a stronger experiment and commit a new record rather than editing the old observation.

Use the journal to protect decisions that do not belong in the book. The journal can explain why a dependency was deferred, why an optional model was not downloaded, why a page layout is interim, or which release input is still user-owned. It should point to benchmark and research files rather than become a second manuscript. Before each commit, update it with the decision, files, commands, result, limitation, and next task. This makes a later session resumable without forcing readers of the book through project-management notes.

Finally, maintain the difference between a learning beta and publication. Passing tests, a site build, and an interim DOCX render show that a source revision is healthy enough to review. They do not choose a GitHub destination, publish a site, export the final Word PDF, choose an ISBN, create a cover, or approve a Lulu proof. Treat those as visible handoffs. The engineering work is to prepare reproducible inputs and honest checks; the release decision remains with the person responsible for the book.

Choosing the next experiment

After the sprint, let a recorded uncertainty choose the next experiment. If a training result is confusing, reduce the data and inspect the gradient or label contract. If a framework result differs, hold the fixture still and compare preprocessing and objective semantics. If retrieval returns a misleading neighbor, preserve the query and excerpt, then decide whether chunking, corpus coverage, or evaluation labels need attention. If a plan is unsupported, retrieve again or stop; do not ask for more fluent prose. If an approval thread becomes stale, start a new scoped proposal rather than reusing an old decision.

This rule prevents a project from becoming a tour of tools. The next library, model, or workflow is justified only when it addresses a named limitation in the current evidence trail. That keeps the manuscript coherent, keeps optional dependencies optional, and gives a reader a repeatable way to continue beyond the first ten days without abandoning the project's central promise: learn modern AI by building systems whose claims, failures, and boundaries can be inspected.

When work pauses, leave a handoff that someone else can use without guessing: the current objective, completed evidence, exact commands that passed, the remaining uncertainty, generated artifacts to ignore, and the next smallest safe action. This is as important for a solo learning project as it is for a team. It turns a future return to the repository into continuation rather than rediscovery.

Use the same standard when a result is disappointing. A failed download, unavailable accelerator, weak retrieval result, malformed structured response, or rejected plan may be the most educational output of the day. Preserve its conditions, state what the system refused to do, and decide whether the durable lesson belongs in a test, fixture, benchmark record, journal, or chapter. A project that records these paths honestly becomes easier to trust and easier to extend than one that presents only successful screenshots.

The final habit is to revisit earlier assumptions after each new chapter. A later retrieval experiment can refine how you describe embeddings; a benchmark can narrow an earlier device observation; a rejected approval can clarify a workflow boundary. Revision is not evidence that the first lesson failed. It is the normal process by which a living technical book stays aligned with the code and records it asks readers to inspect.

Reproducible Publishing Protocol

A print file is not a snapshot of whatever happened to be open on one laptop. It is a build result with inputs, tools, decisions, and review evidence. This appendix defines the small publishing protocol used for this book. It is useful for a first proof, but it becomes essential after an editor, a proofreader, or a printer finding changes the manuscript.

What must be repeatable

There are four things to reproduce: the prose, the layout, the visual assets, and the decision record. Markdown is the prose source. The committed Word reference template is the layout authority. The cover artwork and its editable metadata are visual inputs. The journal records why a particular release was built and what review results mean.

Those layers answer different questions. A Git commit can prove which words were used, but it cannot prove that a font embedded correctly. A PDF preflight can prove page boxes and detect encryption, but it cannot decide whether a diagram is understandable at print scale. A rendered-page review can expose a widow or a clipped caption, but it cannot establish whether an ISBN belongs to the work. Treating one check as a substitute for the others creates quiet, expensive errors.

The release manifest closes part of that gap. It records hashes of the manuscript metadata, front matter, Word template, cover metadata, and approved art plate. If a later PDF differs, compare those hashes before guessing at what changed. A different hash is not an error; it is a prompt to record an edition decision.

The build sequence

Start from a clean working tree. Run the editorial audit before creating a document because a visually attractive PDF with a broken code link is still a broken book. Then build the DOCX, normalize the non-bleed interior to the 6×9 trim, and render it to pages. Use the same order every time:

```
uv run ruff check .
uv run pytest
make audit-book
```

```
make validate-reader-bridge
make qrcodes
make book
make master-pdf
make preflight
make validate-publication
make release-manifest
```

The reproducible route produces one submission candidate at book/build/pdf-online.pdf. It is both the online reading edition and the interior artifact: visual title page, copyright/ISBN page, dedication, generated contents, and manuscript. The final stage embeds the fonts used in generated pages, uses the versioned 300 ppi title asset, flattens only pages with transparency at 300 ppi, and rejects a PDF with residual transparency or unembedded used fonts. The approved ClineFlow visual remains intentionally dark; Lulu may retain its Color Standard ink-coverage warning, which is reviewed on the physical proof.

The reader bridge is one synchronized publishing unit. Its manifest maps every numbered chapter to the live main-branch lab URL, code command, expected result, benchmark, and QR image. The print build derives each chapter-end lab panel from that manifest; the course build derives its lab pages from the same source. Update prose, command, benchmark, manifest, QR, Pages output, and PDF in the same publication change.

After LibreOffice renders the body, the publishing step creates the one master sequence: shared visual title page, copyright/ISBN page, dedication, contents, then the remaining manuscript. It locates chapter starts in that same result, generates the contents from measured pagination, applies folios only after the front matter, and records hashes and page counts in the publication manifest. It then performs the font, transparency, and title-art audit on exactly the PDF submitted to Lulu; page numbers are never maintained by hand.

The build script handles a subtle but important source-sharing detail. Chapter files carry Docusaurus front matter for course navigation. Pandoc would treat that same front matter as document metadata, allowing a chapter title to silently replace the book title. The print build therefore makes temporary copies with that front matter removed; it never edits the canonical chapters. The web and print readers still receive the same prose.

Inspecting the interior

Review pages in a sequence rather than scrolling randomly. Begin with the shared visual title page, copyright, dedication page, and contents page. Confirm that the assigned ISBN matches the centralized Lulu distribution metadata and that there is no accidental barcode in the interior. The barcode belongs only in Lulu's final cover-template area. Then inspect every chapter opener: it should have a clear title, breathing room, and a stable relationship to the preceding chapter. Inspect every chapter ending for an isolated takeaway heading, a lone line, or a next-step reference that points nowhere.

Next inspect the technical pages. Code should remain readable without forcing the reader to rotate the book. Figure captions must stay with their figures. Tables should not extend past the safe area. Citations should retain their brackets and bibliography references. A high-resolution source image can still look bad if Word scaled it badly, so judge the placed result rather than only the file's pixel dimensions.

Finally inspect the first and last printed pages. The first interior page is on the right-hand side of a bound book, so front matter and chapter breaks should respect odd/even sequencing. The final page should end deliberately, without a half-empty bibliography fragment or a dangling heading. A small correction is often enough: move a figure, revise a paragraph, or add a meaningful cross-reference. Do not add filler just to solve a page-turn problem.

Cover boundaries

The provisional cover is a front-cover review asset. It uses a 6.25×9.25 inch full-bleed canvas, an original Rottweiler art plate, and vector title text. The art direction can be approved before the manuscript is frozen, but the final cover cannot. Lulu calculates the wrap width from the actual interior page count and provides a template with the spine and barcode safe zones. A guessed spine is a promise to trim incorrectly.

Keep two cover decisions separate. The first is editorial: is the title, subtitle, illustration, author credit, and imprint communicating the book? The second is production: does the final one-piece back/spine/front PDF match the exact Lulu template, include safe margins and bleed, and contain the assigned ISBN barcode in Lulu's designated safe area? The first can be repeated locally. The second begins after the page count is frozen.

Do not reuse the provisional cover's placeholder metadata or make a barcode outside Lulu's final template. The assigned ISBN is recorded in book/lulu-distribution.yaml and printed as text in the interior; once Lulu provides the final wrap template, regenerate the cover from that same metadata, regenerate the manifest, and visually compare the results before upload.

Handling a proof finding

Every proof finding gets one of four labels: content correction, layout correction, production correction, or deferred decision. A content correction changes canonical Markdown and may require source or experiment review. A layout correction changes styles, the reference template, or build rules. A production correction changes only release inputs such as an image profile or cover template. A deferred decision is an intentional pause, such as choosing an ISBN or approving a cover after a physical proof.

Record the label, affected pages, source files, and verification command in the journal before committing. This makes a second proof explainable. If a page number moved, the journal should show whether it moved because prose changed, a figure changed, or a style changed. The goal is not bureaucratic paperwork; it is to avoid "fixing" a symptom while reintroducing the original problem.

Human approvals are not optional

Automation can build and inspect artifacts, but it cannot make publication decisions. A human must approve the final text, any substantive visual change, the real ISBN and metadata, the Lulu-selected product options, the upload, and the proof order. The same principle appears in the Book Intelligence Assistant: retrieval and critique can prepare evidence, while write and external actions remain behind an approval boundary.

This division keeps the workflow fast without pretending that it is autonomous. Build tools should make the next decision easier to inspect. They should never obscure who made it.

Release checklist

Before calling a PDF release-ready, answer yes to each question:

Is the working tree clean apart from intentionally untracked build outputs?

Do tests, audit, site build, and DOCX build pass from the documented setup?

Does the final PDF use Word rather than the provisional LibreOffice route?

Does preflight confirm the intended page size, single-page layout, and no security protection?

Were fonts, image placement, safety margins, first/last pages, and every chapter opener inspected at print scale?

Is the interior page count frozen before requesting the Lulu cover template?

Do title, author, imprint, ISBN, and barcode agree across the interior, cover, Lulu form, and release manifest?

Has a human approved the physical or digital proof?

If any answer is no, the project is still a beta or a review artifact. That is not a failure. It is an accurate description of the state of the work.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press.

Google Cloud. 2026. "Introducing the Open Knowledge Format." https://cloud.google.com/blog/products/data-analytics/how-the-open-knowledge-format-can-improve-data-sharing.X

Hugging Face. 2026. "Transformers Installation and Pipeline Documentation." https://huggingface.co/docs/transformers/en/installation.X

LangChain. 2026a. "ChatOpenAI Integration." https://docs.langchain.com/oss/python/integrations/chat/openai.X

LangChain. 2026b. "Interrupts." https://docs.langchain.com/oss/python/langgraph/interrupts.X

LangChain. 2026c. "Persistence." https://docs.langchain.com/oss/python/langgraph/persistence.X

MLX Contributors. 2026. "MLX LM." https://pypi.org/project/mlx-lm/.X

Reimers, Nils, and Iryna Gurevych. 2019. "Sentence-BERT: Sentence Embeddings Using Siamese BERT-Networks." *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. https://aclanthology.org/D19-1410/.X

TensorFlow. 2026. "Install TensorFlow with Pip." https://www.tensorflow.org/install/pip.X

Uriostegui, Hassan. 2026a. “ClineFlow: Persistent Context, Open Knowledge.” https://github.com/hassanvfx/clineflow.X

Uriostegui, Hassan. 2026b. “How Lyrics Arrangement Studio Turns Raw Lyrics into Performance-Ready Songs.” https://uriostegui.medium.com/turn-raw-lyrics-into-performance-ready-songs-92f62 322ff9e.X

Uriostegui, Hassan. 2026c. “Lyrics Arrangement Studio.” https://github.com/hassanvfx/lyrics-refiner.X

Uriostegui, Hassan. 2026d. “Newsmusic: News to YouTube Video Generator.” https://github.com/hassanvfx/newsmusic.X

Uriostegui, Hassan. 2026e. “Turn Trending News into YouTube Music Videos.” https://uriostegui.medium.com/75 427bae1309.X

Vaswani, Ashish et al. 2017. “Attention Is All You Need.” *Advances in Neural Information Processing Systems*.

Wolf, Thomas, Lysandre Debut, Victor Sanh, et al. 2020. “Transformers: State-of-the-Art Natural Language Processing.” *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*.