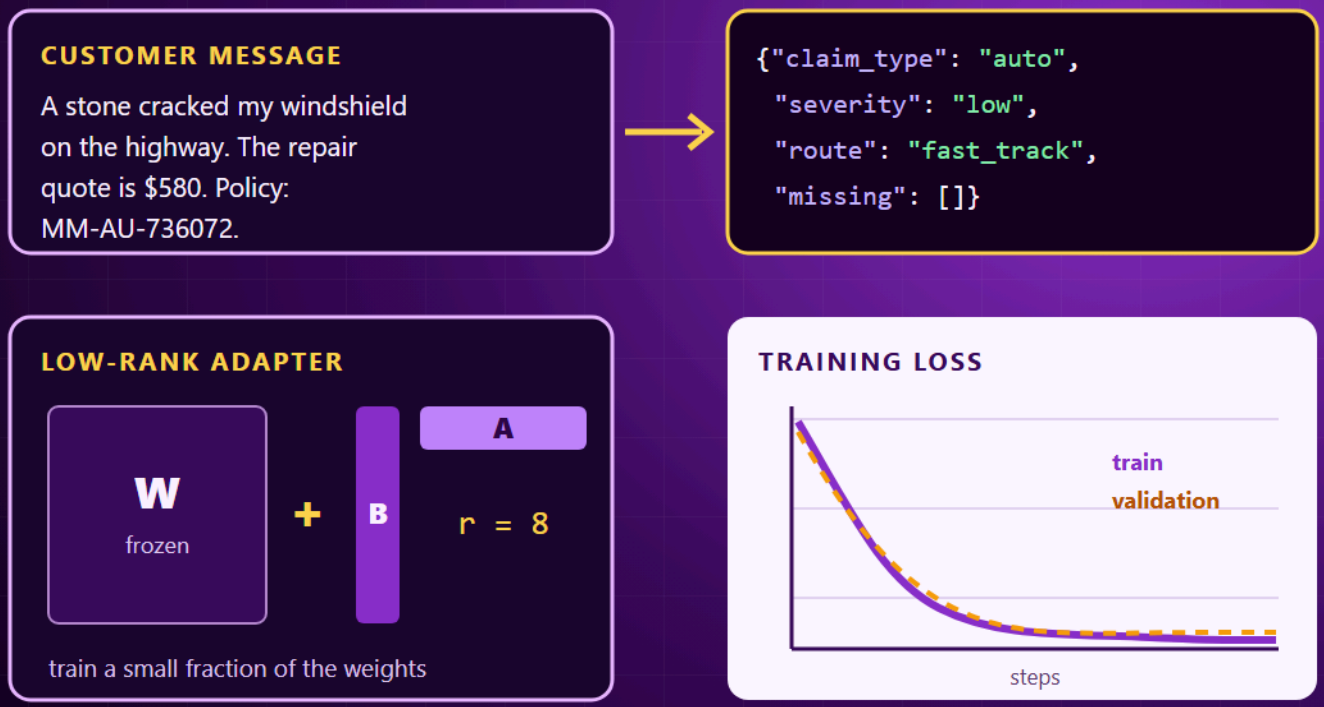



```
config = LoraConfig(r=8, lora_alpha=16, task_type="CAUSAL_LM")
model = get_peft_model(base, config)
trainer = DPOTrainer(model=model, args=args, train_dataset=prefs)
# evaluate before and after, on the same held-out test set
```

A HANDS-ON, PRACTICAL GUIDE

Fine-Tuning Open LLMs

Adapt small open models with PyTorch, PEFT, and TRL — from dataset to deployment

Data · Chat Templates · Full Fine-Tuning · LoRA · QLoRA
SFT · DPO · Evaluation · Red-Teaming · Export · Serving

DATA · TRAINING · ALIGNMENT · EVALUATION · SHIPPING

THENEXBOOKS

Fine-Tuning Open LLMs

Adapt Small Open Models with PyTorch, PEFT, and TRL — from Dataset to Deployment
Data, full fine-tuning, LoRA, SFT, DPO, evaluation, safety, and serving — built step by step

First Edition

THENEXBOOKS

Fine-Tuning Open LLMs
Adapt Small Open Models with PyTorch, PEFT, and TRL — from Dataset to Deployment

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Machine learning libraries such as PyTorch, Transformers, PEFT, TRL, and Datasets change quickly; argument names, defaults, and outputs can differ between versions. Training results depend on hardware, library versions, and random seeds, so your numbers may differ slightly from the ones printed here. Open models are published under their own licenses and acceptable-use terms; always read the license of every model and dataset you use before putting it into production. Nothing in this book is legal, insurance, or compliance advice.

Python is a trademark of the Python Software Foundation. PyTorch is a trademark of The Linux Foundation. Hugging Face is a trademark of Hugging Face, Inc. NVIDIA and CUDA are trademarks of NVIDIA Corporation. Windows is a trademark of the Microsoft group of companies. Other product and model names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by any of these companies or model publishers.

Marrowfield Mutual, the insurance company used as the running example in this book, is fictional, as are its staff, customers, policies, claims, and all sample data. The claims messages are generated by a seeded script printed in the book; phone numbers use the 555-01xx range and email addresses use reserved example domains.

Preface

Open language models have become small enough to run on a laptop and good enough to be useful. A model with a few hundred million parameters can follow instructions, write JSON, and answer questions about a domain it has seen before. Yet the first time you point one at your own business problem, the results are usually disappointing: it ignores the output format, invents fields, misses the rules your team cares about, and answers in a chatty tone nobody asked for.

Fine-tuning is how you close that gap. By training an open model on a few hundred well-chosen examples of the exact task, you can teach it your format, your vocabulary, and your decision rules, and then run it on hardware you control. Done badly, fine-tuning burns time and money and produces a model that looks good on a demo and fails on real traffic. Done well, it gives you a small, cheap, private model that does one job reliably.

This book teaches the whole process, from deciding whether to fine-tune at all to serving the finished model behind an API. We do not stop at calling a library function. We build one real project step by step, write the training loop ourselves before we let a library hide it, measure every change against a fixed test set, and explain *why* each technique works, so you can apply it to your own data and models.

Who This Book Is For

- **Python developers** who have used a hosted AI model through an API and now want to adapt an open model to their own task.
- **Data scientists and ML engineers** who know classic machine learning and want a practical, honest path into training and evaluating language models.
- **Technical leads and architects** who must decide between prompting, retrieval, and fine-tuning, and need to understand costs, risks, and licensing.
- **Students and self-learners** who want to understand how instruction tuning, LoRA, and preference optimization actually work, with code that runs on an ordinary computer.

You should be comfortable with basic Python: functions, lists and dictionaries, and running scripts from a terminal. No prior PyTorch or deep learning experience is required; Part I teaches what you need.

What You Will Learn

- When fine-tuning beats prompting or retrieval-augmented generation, and when it does not.
- How language models learn: tokens, logits, cross-entropy loss, gradients, and why we mask the prompt out of the loss.
- How chat templates and special tokens work, and why getting them wrong silently ruins a fine-tune.

- How to design a task, build a dataset from synthetic data and real logs, clean it, redact personal data, and split it without leakage.
- How to run a full fine-tune in plain PyTorch, then the same job with the Trainer API, and which hyperparameters really matter.
- How LoRA adapters work, how to train them with PEFT, and how QLoRA fits larger models on a single GPU.
- How to use TRL for supervised fine-tuning and Direct Preference Optimization, and where newer methods fit.
- How to evaluate before and after, debug overfitting and forgetting, and red-team a model for safety.
- How to merge adapters, save safetensors, quantize, export, serve the model with FastAPI, and govern it responsibly.

How This Book Is Organized

Part I — Foundations (Chapters 1–5) explains when to fine-tune, sets up the tools, shows how models learn, covers chat templates and generation, and measures a baseline. **Part II — Data** (Chapters 6–10) designs the task and builds, cleans, tokenizes, and versions the dataset. **Part III — Training** (Chapters 11–15) covers full fine-tuning in PyTorch and with the Trainer API, hyperparameters, LoRA, and QLoRA. **Part IV — Alignment with TRL** (Chapters 16–19) covers supervised fine-tuning, preference data, DPO, and other preference methods. **Part V — Evaluation and Safety** (Chapters 20–23) covers metrics, judging, debugging, and red-teaming. **Part VI — Shipping** (Chapters 24–28) covers merging and saving, quantization and export, serving, cost and licensing, and a complete case study. The appendices provide a cheat sheet, templates and reference tables, a production checklist, and a glossary.

Each chapter ends with exercises and a **Key Takeaways** box for quick review.

The Running Example: Marrowfield Mutual

Throughout the book we work with **Marrowfield Mutual**, a fictional mid-sized insurer that sells auto, home, travel, and pet policies. Every day its claims team receives first notice of loss messages from customers through a web form, email, and chat: a cracked windshield, a burst pipe, a lost suitcase, a dog that swallowed a sock. Before an adjuster can help, someone must read each message and decide the claim type, how severe it is, where it should be routed, and what information is missing. Marrowfield wants a small open model that does this triage step, runs on its own servers, and never sends customer messages to an outside service.

We build that **claims triage assistant** from scratch. A seeded script generates 300 synthetic customer messages with known correct answers, so every number in the book can be reproduced. We measure an untuned model, clean and format the data, fine-tune a small open model with a full fine-

tune and with LoRA, add preference tuning with DPO, evaluate and red-team the results, and finish with a merged, quantized model served behind an API.

Versions and Conventions

The code targets **Python 3.12** and recent versions of **PyTorch**, **Transformers**, **Datasets**, **PEFT**, and **TRL**. These libraries move quickly; where an argument name or default has changed between releases, a Note says so. The examples were run on an ordinary Windows PC with a CPU-only PyTorch build, using a deliberately tiny open model (`HuggingFaceTB/SmolLM2-135M-Instruct`) and short training runs, so you can follow along without a GPU. Code that needs an NVIDIA GPU, such as 4-bit QLoRA training, is clearly labeled, and its output is described rather than reproduced. When we refer to a hosted commercial chat model, we write *[your AI model]*.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version differences, or a useful aside.

> **WARNING**
> A common mistake or something that can cause wasted training runs, data leaks, or security problems.

Code, file names, and commands appear in `monospace`; long lines are wrapped to fit the page. The project lives in a folder called `mm-claims` with a Python package named `mm_claims`; each module is introduced in the chapter that needs it and summarized in Appendix B. Commands are shown for Windows (`py -3.12 -m ...`); on macOS and Linux use `python3 -m ...` instead.

Contents

Preface

PART I — FOUNDATIONS

1 Why Fine-Tune? Prompting, RAG, and Fine-Tuning Compared

2 Setting Up Your Workbench: Python, PyTorch, and Small Open Models

3 How Language Models Learn: Tokens, Logits, Loss, and Gradients

4 Chat Templates, Special Tokens, and Generation Settings

5 Baselines First: Measuring the Untuned Model

PART II — DATA

6 Designing the Task: Output Formats, Labels, and Acceptance Criteria

7 Building a Training Set: Synthetic Data, Real Logs, and Annotation

8 Cleaning and Validating Data: Duplicates, PII, Leakage, and Splits

9 From Records to Tensors: Tokenization, Loss Masking, Padding, and Packing

10 Working with Hugging Face Datasets: Loading, Mapping, and Versioning

PART III — TRAINING

11 A Full Fine-Tune in Plain PyTorch

12 The Trainer API: Arguments, Callbacks, Checkpoints, and Logging

13 Hyperparameters That Matter: Learning Rate, Batch Size, Epochs, and Schedules

14 LoRA with PEFT: Low-Rank Adapters Explained and Applied

15 QLoRA and Memory Math: Training Bigger Models on One GPU

PART IV — ALIGNMENT WITH TRL

16 Supervised Fine-Tuning with TRL's SFTTrainer

17 Preference Data: Chosen, Rejected, and How to Collect Them

18 Direct Preference Optimization (DPO) in Practice

19 Beyond DPO: KTO, ORPO, Reward Models, and RL-Based Tuning

PART V — EVALUATION AND SAFETY

20 Evaluating a Fine-Tuned Model: Held-Out Sets, Field Accuracy, and Error Analysis

21 Beyond Exact Match: Rubrics, Model-Based Judges, and Human Review

22 Debugging Fine-Tunes: Overfitting, Forgetting, and Data Bugs

23 Safety, Bias, and Red-Teaming a Fine-Tuned Model

PART VI — SHIPPING

24 Merging Adapters and Saving Models: safetensors and Model Cards

25 Export and Quantization: int8, ONNX, and GGUF

26 Serving Your Model: FastAPI, Batching, and Inference Servers

27 Cost, Licensing, and Governance of Open Models

28 Case Study: Shipping the Marrowfield Claims Assistant

APPENDICES

A Fine-Tuning Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

Foundations

When fine-tuning is the right tool and when prompting or retrieval is better, a working Python and PyTorch setup with small open models, how a language model actually learns from text, chat templates and generation settings, and an honest baseline for the untuned model.

1 · Why Fine-Tune?
2 · Setting Up Your Workbench
3 · How Language Models Learn
4 · Chat Templates and Generation
5 · Baselines First

CHAPTER 1 · PART I — FOUNDATIONS

Why Fine-Tune? Prompting, RAG, and Fine-Tuning Compared

Fine-tuning is the most powerful and the most expensive way to change what a language model does. In this chapter we compare it honestly with prompting and retrieval, meet the insurer whose claims assistant we will build, and see a small open model go from rambling to structured JSON after a few minutes of training on a laptop CPU.

What a Pretrained Model Already Knows

Every open model you can download today went through at least one long, expensive training phase before it reached you. In *pretraining*, the model read an enormous amount of text and learned a single skill: given some tokens, predict the next one. That simple objective forces it to absorb grammar, facts, common reasoning patterns, code, and a great deal of style. A pretrained model is a remarkably good autocomplete engine, but it is not yet an assistant. Ask it a question and it may continue with three more questions, because that is what text on the internet often does.

Most models we actually use went through a second phase, usually called *instruction tuning* or *alignment*. The model is trained on conversations in which a user asks for something and an assistant answers helpfully, and often on preference data in which people (or other models) said which of two answers was better. The model card of the small model we use throughout this book, `HuggingFaceTB/SmolLM2-135M-Instruct`, says exactly this: the instruct version was produced by supervised fine-tuning on chat data and then by Direct Preference Optimization. In other words, the "Instruct" model you download is *already a fine-tuned model*. When we fine-tune it again for our own task, we are doing the same kind of thing its creators did, just on a much narrower problem and with far less data.

That perspective matters. Fine-tuning is not an exotic research technique. It is the ordinary way that general models become specific ones. The questions worth asking are when it is the right tool, what it costs, and how to do it without fooling yourself about the results. The rest of this book answers those questions with code you can run.

Three Ways to Change a Model's Behavior

Suppose a general-purpose model does not do what you need. You have three broad levers. They differ in what they change, how fast you can iterate, and what can go wrong.

```
               what changes                        when it happens
Prompting      the text you send (instructions,    every request
               examples, constraints)
RAG            the text you send, filled with      every request (a search runs
               documents retrieved for this one     before the model is called)
               question
Fine-tuning    the model's weights                 once, offline, before deployment
                                                   (then the new weights are reused)
```

Prompting: change the instructions

Prompting means writing better input. You describe the task, set constraints ("reply with JSON only"), and often include a few worked examples, which is called *few-shot* prompting. Nothing about the model changes; you are steering a fixed function with a cleverer argument. Prompting is fast to try, cheap to change, and easy to roll back. Its limits show up when the instructions get long, when the model keeps drifting from the format, or when the model is simply too small to follow complicated directions. Every token of instruction is also paid for on every single request, in latency and in compute.

RAG: change the context

Retrieval-augmented generation, or RAG, adds a search step. Before calling the model, your application looks up relevant documents (a policy wording, a product manual, last week's price list) and pastes them into the prompt. The model then answers using that fresh material. RAG is the right tool when the knowledge the model needs changes often, is too large to fit in any model, or must be cited back to a source. Updating a document store takes minutes; retraining a model to learn a new fact is slow and unreliable. RAG does not, however, teach the model a new *behavior*. If the model cannot reliably produce your output format, retrieving more documents will not fix that.

Fine-tuning: change the weights

Fine-tuning continues training an existing model on examples of exactly the behavior you want: these inputs should produce these outputs. The optimizer nudges millions of weights so that the right outputs become more likely. After training, the behavior is built in. You no longer need a page of instructions or a dozen examples in every prompt, because the model has seen hundreds of examples during training. That is the big win: consistent format, consistent style, and task skill that a short prompt cannot deliver, at a lower cost per request.

The price is paid up front and continuously. You need a dataset of good examples, a way to measure quality, a training setup, and a plan for retraining when requirements change. A fine-tuned model is also a new artifact that you are responsible for: you must version it, store it, evaluate it, and eventually retire it.

> **NOTE**
>
> The three techniques are not rivals. Many production systems combine them: a fine-tuned model that reliably follows an output format, a short system prompt that sets the context, and a retrieval step that supplies facts the model could not possibly know. This book focuses on the fine-tuning part, but Chapter 5 starts by measuring how far prompting alone gets us, because that number is the bar fine-tuning has to clear.

A Decision Table

When a team asks "should we fine-tune?", the honest answer depends on what is wrong with the current behavior. The table below is the rule of thumb we use throughout the book.

The problem you see	Try first	Fine-tuning helps?
Model does not know your current prices, policies, or documents	RAG	Rarely; facts change faster than you can retrain
Answers must cite a source document	RAG	No; citations come from retrieval
Output format drifts (extra prose, wrong keys, broken JSON)	Prompting, then fine-tuning	Yes; this is its strongest use
Tone or style is wrong for your brand	Prompting	Yes, when prompts get long or inconsistent
A narrow classification or extraction task, high volume	Few-shot prompting	Yes; a small tuned model can replace a large prompted one
The model must refuse or handle certain requests your way	Prompting plus code checks	Often, with preference tuning (Part IV)
Data must never leave your servers	A local open model	Yes, if the local model is too weak without it

Two patterns stand out. First, if the problem is *knowledge*, fine-tuning is usually the wrong tool. A model that has been trained on your 2025 rate table will confidently quote it in 2027. Second, if the problem is *behavior*, especially the consistent production of a specific structure, fine-tuning shines. Our running example is squarely in the second category.

> **TIP**
>
> Before you plan any training, write down three to five real inputs where the current system fails, and describe the output you wanted. If you cannot write the desired outputs by hand, you are not ready to fine-tune, because a training set is nothing more than many such pairs.

What Fine-Tuning Is Good At, and What It Is Not

It helps to be concrete about what changes inside a model when we fine-tune it. Training adjusts the weights so that the probability of our example outputs, given our example inputs, goes up. Anything that can be expressed as "after inputs like this, produce text like that" is a good candidate. In practice that covers:

- **Format.** Always answer with a JSON object with these four keys, in this order, and nothing else.
- **Style and voice.** Write replies the way your support team writes them, at the length they use.
- **Narrow skills.** Classify, extract, normalize, or route text according to rules your examples demonstrate consistently.
- **Compression of instructions.** Behavior that needed a 2,000-token prompt can often be learned so the prompt shrinks to one line, which speeds up every request.
- **Smaller models doing a big model's job.** On a narrow task, a small tuned model can match a much larger prompted model, which matters when you must run on your own hardware.

Fine-tuning is weak or risky for other goals:

- **Injecting facts.** Models can memorize facts from training data, but unreliably, and they do not know when the facts become stale.
- **Fixing reasoning a model fundamentally lacks.** A tiny model will not learn multi-step arithmetic from a few hundred examples. We will see exactly this limit when our model has to compare dollar amounts with thresholds.
- **Guaranteeing rules.** A model makes probabilistic predictions. If a business rule must *always* hold, enforce it in code and use the model for the parts that need language understanding.

The Costs and Risks You Sign Up For

Teams often underestimate fine-tuning because the training command itself is short. The real work sits around it. Here is the list we keep in front of us when we scope a project.

Data work

You need examples that are correct, consistent, representative of real traffic, legally usable, and free of personal data you are not allowed to keep. Part II of this book is devoted to this, because data quality controls the ceiling of everything that follows. Writing annotation guidelines, labeling, reviewing, cleaning, and splitting data typically take longer than training.

Compute and memory

Training needs far more memory than running a model. As a rule of thumb, a full fine-tune with the popular AdamW optimizer in mixed precision needs roughly 16 bytes per parameter before you count activations. For our 135-million-parameter model that is about 2.2 GB, which a laptop handles. For a

7-billion-parameter model it is about 112 GB, which is why techniques like LoRA and QLoRA exist (Chapters 14 and 15). These are approximations, but they are good enough to decide whether a plan is realistic.

Evaluation

A fine-tuned model can look excellent on a few hand-picked examples and still fail systematically. You need a held-out test set, metrics that match the business goal, and the discipline to look at the errors. Chapters 5, 20, and 21 build that machinery.

Forgetting and side effects

Training on a narrow task can erode general abilities, an effect called *catastrophic forgetting*. For a single-purpose assistant that is often acceptable, but you should know it is happening. Chapter 22 shows how to detect it.

Ownership

Once you ship a fine-tuned model, you own a new artifact. You must keep track of which data and code produced it, which license it inherits from its base model, who approved it, and how to roll it back. Chapters 24 and 27 cover model cards, licensing, and governance.

> **WARNING**
>
> Fine-tuning can make a model confidently wrong in a very consistent format. A model that always produces perfectly valid JSON looks trustworthy in a demo even when half of the field values are wrong. Never judge a fine-tune by whether its output *looks* right; judge it by field-level accuracy on data it has never seen.

Open Models Versus Hosted APIs

You can fine-tune two kinds of models. Hosted providers let you upload training data and train a private version of their model through an API; you never see the weights. Open models, also called open-weight models, let you download the weights and train them on your own hardware or in a cloud account you control. This book is about the second kind, for reasons that the running example makes concrete.

Aspect	Hosted fine-tuning	Open model, self-hosted
Where data goes	To the provider	Stays on your machines
Who holds the weights	The provider	You
Infrastructure work	Low	You run training and serving
Cost shape	Per token, per training job	Hardware and engineering time
Model choice	The provider's catalog	Any model whose license fits

Lock-in risk	Model may be retired	Weights are yours to keep

Neither column is always better. If your data can leave your network and a hosted model already works, the hosted path may be cheaper in engineering time. If privacy rules, latency, cost at volume, or long-term control matter more, an open model is attractive. "Open" also does not mean "no rules": every model ships with a license, and some licenses include acceptable-use policies or attribution requirements. We check licenses in Chapter 2 and discuss them in depth in Chapter 27.

Meet Marrowfield Mutual

Our running example is a fictional mid-sized insurer, **Marrowfield Mutual**, which sells four policy lines: auto, home, travel, and pet. When something goes wrong, a customer sends a *first notice of loss*, or FNOL: the first message that tells the insurer about an incident. At Marrowfield, these messages arrive through three channels, a web form, email, and a chat widget, and they land in a busy claims inbox. Someone has to read each one and decide what kind of claim it is, how serious it looks, who should handle it, and what information is still missing before work can start. That reading and sorting is called *triage*.

Four people shape the project. You will meet them again throughout the book.

- **Imogen Thackeray**, Head of Claims Operations, owns the business problem. She decides what "good enough to ship" means and signs off on the acceptance criteria.
- **Nadia Castellane**, an ML engineer, builds the model with us. When the book says "we", think of Nadia and yourself at the same keyboard.
- **Harriet Kettering**, a senior claims adjuster, knows how triage is really done. She writes the annotation guidelines and reviews the model's output.
- **Felix Okonjo**, the Data Protection Officer, makes sure customer data is handled lawfully: what may be used for training, how personal details are removed, and how long anything is kept.

Felix sets the first hard constraint before any code is written: claim messages contain names, email addresses, phone numbers, and descriptions of injuries. They must not be sent to an external service. Whatever model Marrowfield uses has to run on Marrowfield's own servers. That one sentence rules out hosted fine-tuning for this project and points us at small open models.

The triage task

The claims triage assistant reads one FNOL message and answers with a JSON object that has exactly four keys, always in the same order:

- `claim_type`: one of `auto`, `home`, `travel`, `pet`, the policy line the incident falls under.
- `severity`: `low`, `medium`, or `high`.
- `route`: `fast_track` (simple, can be settled quickly), `adjuster` (needs a person), or `urgent` (a human must look now).

- `missing`: a list naming what the customer did not tell us, drawn from `policy_number` and `incident_date`.

Harriet's rules are short. If a person was injured, the claim is high severity and urgent, whatever the amount. Otherwise severity follows the stated cost: under $1,000 is low, $1,000 up to $9,999 is medium, $10,000 or more is high, and no amount yet counts as medium. A claim goes to the fast track only when it is low severity and nothing is missing; everything else goes to an adjuster. Chapter 6 turns these rules into a precise specification, edge cases included. Here is the book's main worked example, test claim `MMC-24190`, which arrived through the web form:

```
Good morning, I hit a deer on a country road and the front of the car is smashed.
The date was April 23. I am still waiting for a quote. Thanks, [NAME]
```

The customer's name has already been replaced with `[NAME]` (we build that redaction step in Chapter 8). The correct answer is:

```
{"claim_type": "auto", "severity": "medium", "route": "adjuster",
 "missing": ["policy_number"]}
```

(The model produces this on one line; we wrapped it for print.) It is an auto claim. Nobody was hurt and there is no amount yet, so severity is medium. Medium severity means it cannot be fast-tracked, so it goes to an adjuster. The date is stated, but the policy number is not, so `missing` lists it. A person does this in a few seconds. Our goal is a model that does it in well under a second, consistently, on Marrowfield's own hardware, with a person still reviewing what matters.

Why not just write rules in code? Partly we will, and the book is honest about that. Amount thresholds are easy to code. But real messages say "the vet bill came to $1,820", "my costs come to $9,550", "I am still waiting for a quote", or "it will cost about $930"; they describe a broken wrist on a ski trip or a scratched car door; they mention dates in a dozen ways. Understanding that variety is what language models are good at. The interesting engineering question, which we return to in Chapter 28, is how to split the work between the model and plain code.

A Preview: Before and After

To make the goal tangible, here is a short script that sends the three worked-example claims to two models: the untuned `SmolLM2-135M-Instruct` exactly as downloaded, and the same model after fine-tuning with the LoRA recipe from Chapter 14 (merged into a single model in Chapter 24). You will not have the fine-tuned model yet, so for now just read the output; by the end of Part III you will produce it yourself. The helper functions `load_model`, `generate_answers`, and `parse_answer` are built in Chapter 5.

before_after.py

```
"""Show the same three claims answered by the untuned and the fine-tuned model."""
import sys

from mm_claims.config import BASE_MODEL, DATA_DIR, offline, set_seed
from mm_claims.data_gen import read_jsonl
from mm_claims.evaluate import generate_answers, load_model, parse_answer

sys.stdout.reconfigure(encoding="utf-8")
offline()
set_seed()

FINE_TUNED = sys.argv[1] if len(sys.argv) > 1 else "outputs/merged"
IDS = ["MMC-24190", "MMC-24198", "MMC-24036"]
records = [r for r in read_jsonl(DATA_DIR / "test.jsonl") if r["claim_id"] in IDS]

for name, path in [("untuned", BASE_MODEL), ("fine-tuned", FINE_TUNED)]:
    tokenizer, model = load_model(path)
    answers = generate_answers(model, tokenizer, records)
    print(f"--- {name}")
    for rec, ans in zip(records, answers):
        obj = parse_answer(ans)
        shown = (" / ".join(map(str, obj.values())) if obj
                 else " ".join(ans.split())[:60])
        print(rec["claim_id"], shown)
print("--- gold")
for rec in records:
    print(rec["claim_id"], " / ".join(map(str, rec["label"].values())))
```

Output

```
--- untuned
MMC-24190 [NAME] [NAME] [NAME] [NAME] [NAME] [NAME] [NAME] [NAME] [NAM
MMC-24036 [NAME] [NAME] [NAME] [NAME] [NAME] [NAME] [NAME] [NAME] [NAM
MMC-24198 JSON: { "claim_type": "claim", "severity": "critical", "rout
--- fine-tuned
MMC-24190 auto / medium / adjuster / []
MMC-24036 travel / high / urgent / []
MMC-24198 auto / medium / adjuster / []
--- gold
MMC-24190 auto / medium / adjuster / ['policy_number']
MMC-24036 travel / high / urgent / ['policy_number']
MMC-24198 auto / low / fast_track / []
```

Read this output slowly, because it summarizes the whole book in miniature. The untuned model, even with a clear system prompt that says "Reply with JSON only", mostly echoes the `[NAME]` placeholder over and over. On the windshield claim it does attempt JSON, but invents values such as `"claim"` and `"critical"` that are not in our vocabulary. Across the full 30-row test set, which we measure properly in Chapter 5, only 20% of its answers are valid JSON and not a single field value is correct.

The fine-tuned model always produces clean JSON in the right shape, and it gets the claim type right every time. It recognized the skiing injury as urgent. But look closer: it says nothing is missing on two claims where the policy number is absent, and it called a $290 windshield repair medium severity instead of low. Over the whole test set this model gets every field right on 40% of claims. That is a dramatic jump from zero, and it is also nowhere near good enough to route real claims without a person checking. Both facts are true at once, and learning to hold both is the most important habit in this field.

> **NOTE**
>
> All numbers in this book come from real runs on a laptop-class CPU with fixed random seeds, and our test set has only 30 rows, so one claim is worth 3.3 percentage points. Treat small differences between experiments as noise. We say so every time it matters.

Why We Start Tiny

"Small model" means very different things to different people. In this book it means `HuggingFaceTB/SmolLM2-135M-Instruct`: 134,515,008 parameters, about 270 MB on disk, released under the Apache-2.0 license. For comparison, models people call "small" in production conversations often have billions of parameters. We chose a tiny model deliberately:

- **Everything runs on a CPU.** Every training run in the main text finishes in a few minutes on an ordinary laptop, so you can follow along without renting a GPU. You see real loss curves and real errors, not screenshots.
- **Mistakes are cheap.** Fine-tuning is learned by breaking things: a learning rate that is too high, a mask that hides the wrong tokens, a template mismatch. On a tiny model you find out in minutes.
- **The code is identical at larger scale.** The libraries we use (PyTorch, Transformers, PEFT, TRL) do not care whether the model has 135 million or 7 billion parameters. Changing the model id and the hardware settings is most of the work of scaling up, and Chapter 15 shows the memory math for doing it.
- **Its weaknesses are instructive.** A tiny model struggles with exactly the things that every model struggles with to some degree, such as comparing numbers and noticing what is absent. Seeing those failures clearly prepares you to evaluate larger models honestly.

The table below gives approximate memory needs, which explain why the same project on a 7-billion-parameter model requires very different hardware. Numbers are rough rules of thumb: weights only for the first three columns, and about 16 bytes per parameter for a full fine-tune before activations.

Model	fp32 weights	bf16 weights	4-bit weights	Full fine-tune
SmolLM2-135M	0.54 GB	0.27 GB	~0.07 GB	~2.2 GB
Qwen2.5-0.5B	1.98 GB	0.99 GB	~0.25 GB	~7.9 GB

A 7B-class model	28 GB	14 GB	~3.5 GB	~112 GB

We also use `Qwen/Qwen2.5-0.5B-Instruct`, another Apache-2.0 model, to compare tokenizers and chat templates in Chapter 4, but we do not train it: on a CPU it is simply too slow for a hands-on book. When we show code for larger models or for GPUs, we label it clearly as untested on our machine and describe what you should expect to see.

The Plan of the Book

The chapters follow the order in which a real project unfolds. Each part ends with something that works.

Part	Chapters	What you build
I — Foundations	1–5	A workbench, intuition for tokens and loss, chat formatting, and a measured baseline
II — Data	6–10	A task specification, 300 clean labeled claims, splits, and masked training tensors
III — Training	11–15	Full fine-tunes in plain PyTorch and with Trainer, tuned hyperparameters, LoRA, QLoRA math
IV — Alignment with TRL	16–19	SFT with TRL, preference pairs, DPO, and a map of other methods
V — Evaluation and Safety	20–23	Error analysis, judges and human review, debugging, red-teaming
VI — Shipping	24–28	Merged models, quantization, a FastAPI service, governance, and the final go/no-go decision

All code lives in one project folder called `mm-claims`, organized as a small Python package named `mm_claims`. Each chapter adds or explains one module, and every module can be run on its own with `py -3.12 -m mm_claims.<module>`. Chapter 2 creates the folder and the shared configuration.

How to Read the Code in This Book

A few conventions make the listings easier to follow.

- Commands are written for Windows with the Python launcher, for example `py -3.12 -m mm_claims.evaluate`. On macOS or Linux use `python3 -m mm_claims.evaluate` instead; everything else is the same.
- A label above a listing, such as `mm_claims/formatting.py`, tells you which file the code belongs to. A label reading *Output* shows what we saw when we ran it.
- Long lines in outputs, such as chat-formatted prompts, are sometimes wrapped for print. When that happens we say so.

- Libraries in this space rename arguments regularly. We use the names from recent versions and add a note whenever an older name is still common in tutorials you may find online.
- When a model or tool is not something we could run on our machine (for example GPU-only 4-bit training), the text says so and labels the output as illustrative.

Where the book needs a large commercial assistant, for instance to help draft synthetic data or to judge open-ended answers, we write *[your AI model]*. Use whichever model your organization has approved; nothing in the book depends on a particular vendor.

> **TIP**
>
> Keep a lab notebook from the very first run: date, git commit or folder snapshot, data version, hyperparameters, and the metrics you got. Fine-tuning projects generate many similar-looking experiments, and "which run was the good one?" is a surprisingly common and expensive question.

Try It Yourself

1. Pick a task from your own work that you might want a model to do. Classify each part of it with the decision table: is the difficulty knowledge (RAG), behavior (fine-tuning), or both?
2. Write five input messages for that task and, by hand, the exact output you would want for each. Note how long it takes and where you hesitated. Those hesitations are future annotation guidelines.
3. Using Harriet's rules from this chapter, label claim `MMC-24198` yourself: "A stone cracked my windshield on the highway. This was on September 20. The estimate I got is $290. I am insured under MM-AU-813486." Then check your answer against the gold row in the preview output.
4. Do the same for this message, which has no policy number: "A stone cracked my windshield on the highway. I think it will cost about $370. The date was May 3." Why is the route not `fast_track`, even though the severity is low?
5. List the personal data that might appear in a claim message in your country. Which items would your own data protection officer want removed before training?