Uncodemy · Docs v1.0

Fine-tune small language models with LoRA, from your browser.

LoRA Studio is a full-stack project: upload a dataset, train a LoRA adapter on a small open model, watch the loss fall live, and compare the base and fine-tuned answers side by side. Built with React, Node, Express, MongoDB and a Python trainer.

Try the demo ▶ <> View the code

03	07	01
Services (React, Node, Python)	Tunable training settings	Command to deploy

01 GETTING STARTED

Complete Beginner's Guide

This document explains what LoRA Studio does, how every part is built, how to run it, and how to change it. You do not need machine-learning experience to follow along.

LoRA Studio lets a user teach a small language model a new behavior, for example answering questions in the tone and facts of a company's support team. The user uploads a file of example question and answer pairs, clicks one button, and the system fine-tunes a model, shows progress live, and then lets the user test the result next to the original model.

What is LoRA?

A language model is a very large set of numbers (parameters). Normal fine-tuning changes all of them, which needs a lot of memory. **LoRA (Low-Rank Adaptation)** freezes the original numbers and trains two tiny matrices next to selected layers instead. Only those tiny matrices are saved, so the result (an *adapter*) is small, cheap to train, and can be attached to or removed from the base model at will.

Small adapters · Runs on a laptop CPU · Compare base vs tuned · One-command Docker deploy

> **Who is this for?**
>
> Students learning full-stack and AI engineering, developers who want a working fine-tuning pipeline to study, and teams that want a starting point for a private model-training portal.

---

02 SETUP

Prerequisites & How to Run

The quickest route is Docker: one install, one command.

What you need

- **Docker Desktop** with Compose v2 (on Windows, enable WSL2).
- **8 GB RAM** minimum (16 GB recommended) and about 8 to 10 GB of free disk.
- **Internet on the first run** to download images and the base model (about 270 MB for SmolLM2-135M).
- Free port `8080` (change it with `WEB_PORT`).

How to run it

terminal Copy

```
cp .env.example .env                # then fill JWT_SECRET and TRAINER_SECRET
openssl rand -hex 32                # run twice to generate two secrets
docker compose up --build -d
docker compose ps                   # all four containers should be Up
```

🚀 **Open the app**

Go to `http://localhost:8080`, register an account, then upload `sample-data/support-bot.jsonl` to try a first run.

Without Docker you need Node.js 20+, Python 3.10 to 3.12 and MongoDB. Run the three services in three terminals:

three terminals — Copy

```
# 1) trainer
cd trainer && python -m venv .venv && source .venv/bin/activate
pip install torch --index-url https://download.pytorch.org/whl/cpu
pip install -r requirements.txt
TRAINER_SECRET=dev-only-trainer-secret ADAPTER_DIR=./adapters uvicorn main:app --
port 8000

# 2) backend
cd backend && cp .env.example .env && npm install && npm run dev

# 3) frontend (opens at http://localhost:5173)
cd frontend && npm install && npm run dev
```

⚠️ **Matching secrets**

The trainer and the backend must use the same `TRAINER_SECRET`. If they differ, every training request fails with 401.

Folder structure

lora-studio/ — Copy

```
docker-compose.yml  .env.example  README.md
sample-data/support-bot.jsonl
```

```
trainer/     main.py  training.py  inference.py  common.py  config.py  Dockerfile
backend/     src/server.js  config.js  trainer.js
             src/models/ (User, Dataset, Job)  src/routes/ (auth, datasets, jobs,
inference, internal)
             src/middleware/auth.js  src/utils/parseDataset.js
frontend/    src/App.jsx  api.js  auth.jsx  styles.css
             src/pages/ (Login, Datasets, Jobs, NewJob, JobDetail, Playground)
             Dockerfile  nginx.conf
```

03 OVERVIEW

Features & Stack

Everything a fine-tuning workflow needs, in one small project.

Accounts and isolation

Email and password login with JWT. Every dataset, job and answer belongs to its owner.

Dataset upload

Upload .jsonl, .json or .csv. Rows are cleaned and normalized to instruction, input and output.

LoRA training

Train a small adapter on a base model such as SmolLM2-135M, with seven adjustable settings.

Live progress

A progress bar, loss chart and log lines update every few seconds. Cancel any time.

Playground

Ask one question and see the fine-tuned and base model answers side by side.

Adapter download

Download the trained adapter as a zip and use it anywhere with Hugging Face PEFT.

Tech stack

LAYER	TECHNOLOGY	WHY
Frontend	React 18, Vite, Recharts	Fast build, simple routing, loss chart
API	Node 20, Express 5, Mongoose	Auth, validation, job tracking, secure gateway
Database	MongoDB 7	Users, datasets, jobs, metrics, logs
Trainer	Python 3.11, FastAPI, transformers, PEFT, PyTorch	Only Python has the LoRA tooling
Deploy	Docker Compose, nginx	One command, one public port

04 ARCHITECTURE

How It Works — End-to-End Flow

From a CSV file to a fine-tuned model in six moves.

Upload data
JSONL, JSON or CSV

Create job
Validate settings

Queue
One job at a time

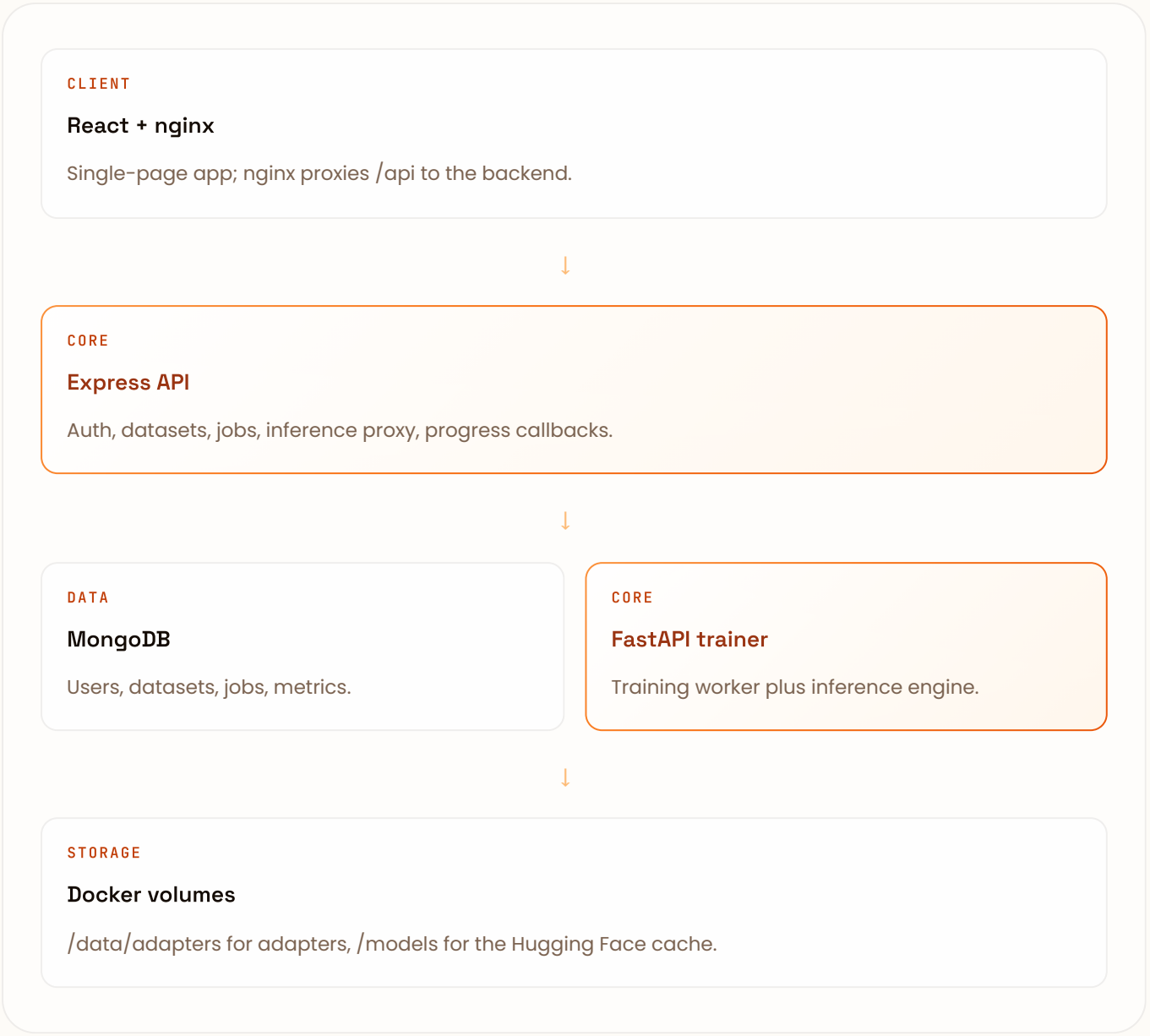
Train
LoRA via PEFT

Save adapter
Stored on a volume

Compare
Base vs tuned

Internal architecture

Node owns everything a web product needs. The Python trainer is a private worker with no public port; every call to it carries a shared secret.



Life of a training job

1. The API validates the request, clamps every setting to a safe range and creates a job with status `queued`.
2. It asks the trainer to start; the trainer queues the job and answers immediately.
3. A single worker thread downloads the dataset, loads the base model and attaches LoRA.
4. At each logging step the trainer posts the loss to the API, which stores it.
5. The browser polls the job every 2.5 seconds and redraws the chart.

6. When training ends the adapter is saved to disk and the job becomes `completed`.

---

05 USAGE

Step-by-Step Usage

A first run takes a few minutes on a laptop.

01 Register

Open the app, choose *No account? Register* and use any email with a password of 8+ characters.

02 Upload a dataset

Go to *Datasets* and upload `sample-data/support-bot.jsonl`. Click *Preview* to see the first five normalized rows.

03 Start a fine-tune

Open *Training jobs*, then *New fine-tune*. Pick the dataset and `SmolLM2-135M-Instruct`. For this tiny dataset set *Epochs* to 10.

04 Watch it train

The job page shows status, a progress bar, the loss chart and logs. The first run also downloads the base model.

05 Open the Playground

When the status is *completed*, click *Try it* and ask a question from your dataset.

06 Compare

The left answer comes from your fine-tuned model, the right one from the untouched base model.

07

Download the adapter

Click *Download adapter* to get a zip you can load with Hugging Face PEFT.

Why 10 epochs?

The sample has 15 rows, so with batch size 4 each epoch is 4 steps. At 3 epochs that is only 12 steps and the change can be subtle. Real projects need hundreds of good examples.

---

06 <> CODE

Understanding the Code, Block by Block

The excerpts below are copied from the real project files.

6.1 — The prompt template

Training and inference must format prompts identically, so the template lives in one file. Different base models use different chat formats, so a simple model-agnostic template is used.

common.py Copy

```
def format_prompt(instruction: str, input_text: str = "") -> str:
    if input_text and input_text.strip():
        return (
            "Below is an instruction paired with an input. Write a response that
completes the request.\n\n"
            f"### Instruction:\n{instruction.strip()}\n\n###
Input:\n{input_text.strip()}\n\n### Response:\n"
        )
    return (
        "Below is an instruction. Write a response that completes the request.\n\n"
        f"### Instruction:\n{instruction.strip()}\n\n### Response:\n"
    )
```

6.2 — Turning rows into training data

Prompt and answer are tokenized separately. The prompt tokens get the label `-100`, which PyTorch ignores, so the model is graded only on the answer. An end-of-text token teaches it when to stop.

training.py

Copy

```
class SFTDataset(Dataset):
    """Instruction tuning data; loss is computed on the response tokens only."""

    def __init__(self, rows, tokenizer, max_len):
        self.items = []
        eos = tokenizer.eos_token_id
        for r in rows:
            p_ids = tokenizer(format_prompt(r["instruction"], r.get("input", "")),
add_special_tokens=False)["input_ids"]
            o_ids = tokenizer(r["output"], add_special_tokens=False)["input_ids"] +
[eos]
            ids = (p_ids + o_ids)[:max_len]
            labels = ([-100] * len(p_ids) + o_ids)[:max_len]
            if any(l != -100 for l in labels):
                self.items.append({"input_ids": ids, "labels": labels})

    def __len__(self):
        return len(self.items)

    def __getitem__(self, i):
        return self.items[i]
```

6.3 — Attaching LoRA

`target_modules="all-linear"` adds adapters to every linear layer, so the same code works for Llama, Qwen and SmolLM style models. Only the adapter weights are trainable.

training.py

Copy

```
lora = LoraConfig(
    r=cfg["loraR"], lora_alpha=cfg["loraAlpha"], lora_dropout=cfg["loraDropout"],
    target_modules="all-linear", bias="none", task_type="CAUSAL_LM",
)
model = get_peft_model(model, lora)
trainable, total = model.get_nb_trainable_parameters()
```

6.4 — Reporting progress and cancelling

A Hugging Face callback posts each logged loss to the API and checks a cancel flag after every step.

training.py

Copy

```
class Reporter(TrainerCallback):
    def __init__(self, job_id, callback_url):
        self.job_id, self.url = job_id, callback_url

    def on_log(self, args, state, control, logs=None, **kw):
        if logs and "loss" in logs:
            report(self.url, step=state.global_step, maxSteps=state.max_steps,
                   epoch=round(state.epoch or 0, 2), loss=float(logs["loss"]))

    def on_step_end(self, args, state, control, **kw):
        if cancel_flags.get(self.job_id):
            control.should_training_stop = True
```

6.5 — Swapping adapters at inference

The base model is loaded once. An adapter is attached when needed, and the base answer is produced with the adapter switched off, so no second copy of the model is required.

inference.py

Copy

```
if adapter_id:
    path = ADAPTER_DIR / adapter_id
    if not path.exists():
        raise FileNotFoundError("Adapter not found")
    if b["peft"] is None:
        b["peft"] = PeftModel.from_pretrained(b["model"], str(path),
adapter_name=adapter_id).eval()
        b["adapters"].add(adapter_id)
    elif adapter_id not in b["adapters"]:
        b["peft"].load_adapter(str(path), adapter_name=adapter_id)
        b["adapters"].add(adapter_id)
    b["peft"].set_adapter(adapter_id)
    out = b["peft"].generate(**inputs, **kwargs)
elif b["peft"] is not None:
    with b["peft"].disable_adapter():
```

```
        out = b["peft"].generate(**inputs, **kwargs)
else:
    out = b["model"].generate(**inputs, **kwargs)
```

6.6 — Validating settings in the API

Whatever the browser sends, each value is clamped to a safe range before a job is created.

jobs.js

Copy

```
const params = {
  epochs: num(b.epochs, 3, 1, 20),
  learningRate: num(b.learningRate, 2e-4, 1e-6, 1e-2),
  batchSize: Math.round(num(b.batchSize, 4, 1, 32)),
  loraR: Math.round(num(b.loraR, 8, 1, 128)),
  loraAlpha: Math.round(num(b.loraAlpha, 16, 1, 512)),
  loraDropout: num(b.loraDropout, 0.05, 0, 0.5),
  maxLength: Math.round(num(b.maxLength, 256, 32, 1024)),
};
```

6.7 — Receiving progress from the trainer

Updates for jobs that already finished are ignored, so a cancelled job stays cancelled.

internal.js

Copy

```
if (TERMINAL.includes(job.status)) return res.json({ ignored: true });

const { status, step, maxSteps, epoch, loss, log, error } = req.body || {};
const set = {};
const push = {};

if (["running", "completed", "failed", "cancelled"].includes(status)) {
  set.status = status;
  if (status === "running" && !job.startedAt) set.startedAt = new Date();
  if (TERMINAL.includes(status)) set.finishedAt = new Date();
}
if (Number.isFinite(step)) set.step = step;
if (Number.isFinite(maxSteps)) set.maxSteps = maxSteps;
if (Number.isFinite(epoch)) set.epoch = epoch;
if (typeof error === "string") set.error = error.slice(0, 2000);
```

```
if (Number.isFinite(loss) && Number.isFinite(step))
  push.metrics = { $each: [{ step, epoch, loss }], $slice: -2000 };
if (typeof log === "string")
```

6.8 — Live updates in React

The job page polls while a job is queued or running and stops once it finishes.

JobDetail.jsx

Copy

```
useEffect(() => {
  load();
  const t = setInterval(() => {
    if (["queued", "running"].includes(statusRef.current)) load();
  }, 2500);
  return () => clearInterval(t);
}, [load]);
```

---

07 CUSTOMIZE

Customization Guide

Common changes and exactly where to make them.

Change the accent color

Every color of the app lives in variables at the top of `frontend/src/styles.css`. Change `--accent` and `--accent-dark`, then rebuild the frontend.

styles.css

Copy

```
:root {
  --bg:#ffffff; --panel:#ffffff; --soft:#fff5f1; --line:#ececec; --text:#1a1a1a; --
muted:#6b7280;
  --accent:#ff5421; --accent-dark:#e8420f; --danger:#d92d20; --ok:#12a150; --
warn:#d97706;
}
```

Add another base model

Set `ALLOWED_MODELS` (comma-separated Hugging Face ids) on the backend. Llama and Qwen style models work with `all-linear`; larger models need more memory.

docker-compose.yml Copy

```
ALLOWED_MODELS: ${ALLOWED_MODELS:-HuggingFaceTB/SmolLM2-135M-
Instruct,HuggingFaceTB/SmolLM2-360M-Instruct,Qwen/Qwen2.5-0.5B-Instruct}
```

Change the default training settings

The form defaults live in `frontend/src/pages/NewJob.jsx`. The server-side fallbacks and limits are in `backend/src/routes/jobs.js`.

NewJob.jsx Copy

```
const defaults = { epochs: 3, learningRate: 0.0002, batchSize: 4, loraR: 8,
loraAlpha: 16, loraDropout: 0.05, maxLength: 256 };
```

Change the prompt template

Edit `trainer/common.py`. Adapters trained with an old template will no longer match, so retrain them.

Train on a GPU

Use `trainer/Dockerfile.gpu` as the trainer image and reserve the GPU in `docker-compose.yml`. This needs the NVIDIA driver and Container Toolkit on the host.

docker-compose.yml Copy

```
trainer:
  build:
    context: ./trainer
    dockerfile: Dockerfile.gpu
  deploy:
    resources:
      reservations:
```

```
      devices:
        - driver: nvidia
          count: 1
          capabilities: [gpu]
```

08 REFERENCE

Beginner Glossary

Terms used throughout the project.

TERM	MEANING
Fine-tuning	Continuing to train a pre-trained model on your own examples to change its behavior.
LoRA	Low-Rank Adaptation: trains small add-on matrices instead of all model weights.
Adapter	The small set of trained LoRA weights that attach to the base model.
Base model	The original pre-trained model, which stays frozen.
PEFT	Hugging Face library for parameter-efficient fine-tuning, including LoRA.
Rank (r)	Size of the LoRA matrices. Higher means more capacity and a larger adapter.
Alpha	Scaling factor for the adapter's contribution. A common choice is 2 times the rank.
Epoch	One full pass over the training data.
Step	One weight update on one batch of examples.
Loss	A number measuring prediction error. Lower usually means a better fit.
Overfitting	Memorizing the training examples instead of learning the pattern.

JWT	JSON Web Token: a signed token that proves who the user is.

09 HELP

FAQ & Troubleshooting

The most common problems and how to fix them.

– The job fails with “Could not reach trainer service”.

The trainer is not running or `TRAINER_URL` is wrong. With Docker run `docker compose logs trainer`. Locally, open `http://localhost:8000/health`.

– The trainer returns 401 or the job never updates.

The secrets differ. Make `TRAINER_SECRET` identical in the backend and the trainer, then restart both.

– The job stays queued.

Only one job runs at a time, so another job may be running. If the trainer restarted, the queue was lost: cancel the job, delete it and start it again.

– The container is killed or exits with code 137.

It ran out of memory. Give Docker more RAM, use a smaller model, or lower the batch size and maximum length.

– The first answer in the Playground is very slow.

The trainer is loading the base model. This happens once per model; later answers are fast.

– The fine-tuned answers look the same as the base model.

There were too few training steps for the dataset. Increase the epochs, add more examples, or raise the rank to 16.

My upload is rejected.

The file needs at least 5 rows with an `instruction` (or prompt / question) and an `output` (or response / answer) field. Maximum size is 5 MB.

How do I reset everything?

Run `docker compose down -v`. This deletes the database, adapters and the model cache.

---

10 NEXT STEPS

Ideas to Extend the Project

Next steps once the basics work.

- **Persistent queue:** use Redis and BullMQ so jobs survive restarts and several trainers share the work.
- **Instant progress:** replace polling with server-sent events or WebSockets.
- **QLoRA:** 4-bit quantization to fine-tune larger models on small GPUs.
- **Evaluation:** add a validation split and show validation loss and sample answers per epoch.
- **Bigger data:** store datasets in GridFS or S3 and import from the Hugging Face Hub.
- **Tests:** Jest and Supertest for the API, Playwright for the UI, and a CI pipeline.

ON THIS PAGE

00 Interactive Demo

01 Introduction

02 Setup & Run

03 Features & Stack
04 How It Works
05 Step-by-Step Usage
06 The Code, Block by Block
07 Customization Guide
08 Glossary
09 FAQ & Troubleshooting
10 Ideas to Extend