

Fine-Tuning LLMs

When and how to customize a pre-trained model on your own data.

Last reviewed 28 Sep 2026 ·

Prerequisites: [LLM APIs](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/) · [Evals](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/)

Related: [Local LLMs](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/) · [MLflow](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/mlflow/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/mlflow/) · [RAG](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/) · [Glossary](https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/) (https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/)

Overview

Challenge: A pre-trained LLM has broad general knowledge but no knowledge of an organization's tone, domain terminology, or required output formats.

Solution: fine-tuning continues training the model on examples specific to the use case. Given hundreds or thousands of (input, ideal output) pairs, the model adjusts its weights to produce outputs closer to those examples.

Pre-trained model:	Knows everything generally
Fine-tuned model:	Knows your specific thing very well

Examples of what fine-tuning fixes:

- "Always respond in SQL, never prose"
- "Use our internal table naming convention"
- "Output JSON that matches our exact schema"
- "Write in our company's brand voice"
- "Classify support tickets into our 40 internal categories"

Fine-tuning compared with RAG:



Use RAG when:

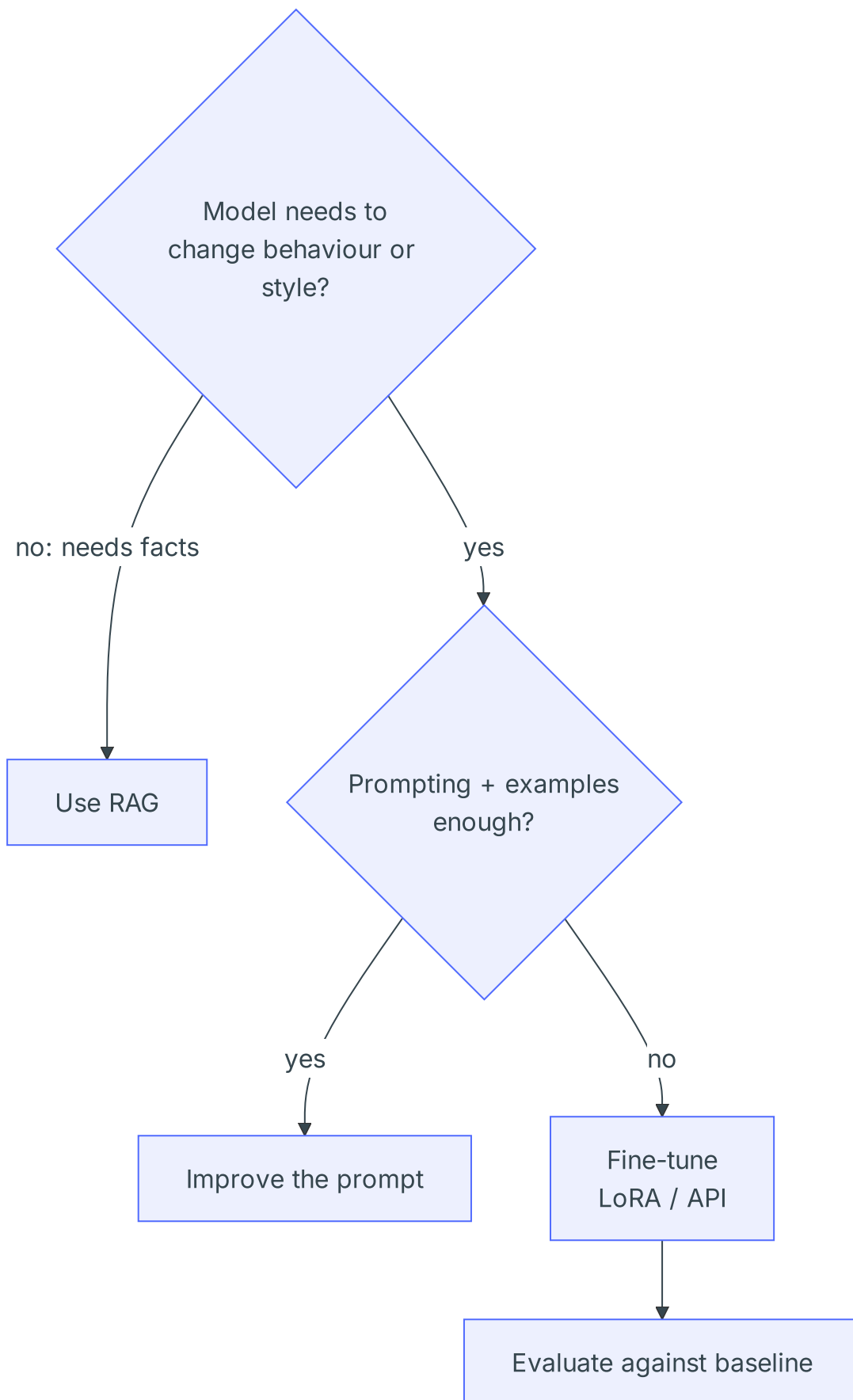
- Your knowledge base changes frequently
- You need source citations
- You want to add new facts the model doesn't know

Use fine-tuning when:

- You need a consistent output format the model ignores in prompts
- You need a specific tone or style the model doesn't adopt
- You're classifying into custom categories not in the base model
- You need faster inference (smaller fine-tuned model > larger base model)
- RAG works but the model still doesn't follow instructions reliably

Use both when:

- Fine-tune for behavior/format, RAG for knowledge



On this page

Basic - [Core Concepts](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/#core-concepts) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/#core-concepts) - [When Fine-Tuning Helps \(and When It Doesn't\)](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/#when-fine-tuning-helps-and-when-it-doesnt) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/#when-fine-tuning-helps-and-when-it-doesnt) -

[Preparing Training Data](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/#preparing-training-data) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/#preparing-training-data)

Intermediate - [Fine-Tuning with OpenAI](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/#fine-tuning-with-openai) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/#fine-tuning-with-openai) - [Fine-Tuning with Hugging Face](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/#fine-tuning-with-hugging-face) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/#fine-tuning-with-hugging-face) - [LoRA / PEFT \(Parameter-Efficient Fine-Tuning\)](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/#lora-peft-parameter-efficient-fine-tuning) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/#lora-peft-parameter-efficient-fine-tuning)

Advanced - [Evaluating Fine-Tuned Models](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/#evaluating-fine-tuned-models) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/#evaluating-fine-tuned-models) - [Dataset Construction Patterns](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/#dataset-construction-patterns) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/#dataset-construction-patterns) -

[Production Considerations](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/#production-considerations) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/#production-considerations) - [Common Pitfalls](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/#common-pitfalls) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/#common-pitfalls)

Reference - [Cheat Sheet](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/#cheat-sheet) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/#cheat-sheet) - [Interview Questions](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/#interview-questions) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/#interview-questions) - [Further Reading](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/#further-reading) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/#further-reading)

Core Concepts

Concept	Description
Full fine-tuning	Update all model weights — most powerful, most expensive, needs multiple large GPUs for a 7B+ model
LoRA	Update only a tiny fraction of weights via low-rank matrices — far cheaper, often close to full fine-tuning quality
PEFT	Parameter-Efficient Fine-Tuning — umbrella term for LoRA and similar techniques
Training data	(prompt, completion) pairs showing the model what good output looks like
Epochs	How many times the model trains over your entire dataset

Concept	Description
Overfitting	Model memorizes training examples instead of learning the pattern — use a validation set
Base model	The starting point — a pre-trained model you fine-tune from
Adapter	A small trained add-on (LoRA) attached to the base model — easy to swap

When Fine-Tuning Helps (and When It Doesn't)

```
# Signs fine-tuning is the right call:
GOOD_CANDIDATES = [
    "Output format: model ignores JSON schema even with
detailed prompts",
    "Style: model writes formally but you need casual/brand
voice",
    "Classification: 40+ custom categories not in base
model's vocabulary",
    "Extraction: model misses domain-specific entities
(internal product names)",
    "Latency: need a smaller, faster model for high-volume
inference",
    "Cost: a small fine-tuned model can cost less per token
than a frontier API model at high volume",
]

# Signs fine-tuning won't help:
BAD_CANDIDATES = [
    "Knowledge: model doesn't know facts that change weekly
→ use RAG",
    "Hallucination: model makes things up → fine-tuning
won't fix grounding",
    "Small dataset: < 50 examples → few-shot prompting is
enough",
    "One-off task: task runs once per day → prompting is
cheaper",
    "No eval: you can't measure if fine-tuning improved
things",
]
```

Preparing Training Data

Training data quality matters far more than quantity. 50 excellent examples beat 5000 mediocre ones.

```
import json
from pathlib import Path

# OpenAI fine-tuning format: JSONL, one example per line
# Each example is a full conversation

def create_training_example(user_msg: str, assistant_msg: str,
                           system: str = None) -> dict:
    messages = []
    if system:
        messages.append({"role": "system", "content": system})
    messages.append({"role": "user", "content": user_msg})
    messages.append({"role": "assistant", "content": assistant_msg})
    return {"messages": messages}

# Example: fine-tune for SQL generation
examples = [
    create_training_example(
        system="You are a SQL expert. Generate ANSI SQL only. No explanations.",
        user_msg="Count orders by status for the last 30 days",
        assistant_msg="""SELECT status, COUNT(*) AS order_count
FROM orders
WHERE created_at >= DATEADD(day, -30, CURRENT_TIMESTAMP())
GROUP BY status
ORDER BY order_count DESC;""")
    ,
    create_training_example(
        system="You are a SQL expert. Generate ANSI SQL only. No explanations.",
```

```

        user_msg="Show me the top 10 customers by revenue
this year",
        assistant_msg="""SELECT c.customer_id, c.name,
SUM(o.amount) AS total_revenue
FROM orders o
JOIN customers c ON o.customer_id = c.id
WHERE YEAR(o.created_at) = YEAR(CURRENT_DATE())
GROUP BY c.customer_id, c.name
ORDER BY total_revenue DESC
LIMIT 10;"""
    ),
    # ... add 50-200 more examples
]

# Split first, so no validation example also appears in the
training file
# (shuffle before splitting if the examples are ordered)
split = int(len(examples) * 0.8)

with open("training_data.jsonl", "w") as f:
    for ex in examples[:split]:
        f.write(json.dumps(ex) + "\n")

# Validation file (10-20% of the data)
with open("validation_data.jsonl", "w") as f:
    for ex in examples[split:]:
        f.write(json.dumps(ex) + "\n")

```

Data quality checklist

```

def validate_training_data(filepath: str) -> dict:
    """Check training data for common problems."""
    issues = []
    examples = []
    total_tokens = 0

    with open(filepath) as f:
        for i, line in enumerate(f):
            try:
                ex = json.loads(line)
                examples.append(ex)

```

```

except json.JSONDecodeError:
    issues.append(f"Line {i}: invalid JSON")
    continue

msgs = ex.get("messages", [])

# Must have user + assistant turn
roles = [m["role"] for m in msgs]
if "user" not in roles:
    issues.append(f"Line {i}: missing user
message")
if "assistant" not in roles:
    issues.append(f"Line {i}: missing assistant
message")

# Estimate token count
text_length = sum(len(m["content"]) for m in
msgs)
total_tokens += text_length // 4 # rough
estimate

avg_tokens = total_tokens // max(len(examples), 1)

return {
    "total_examples": len(examples),
    "issues": issues,
    "avg_tokens": avg_tokens,
    "ready": len(issues) == 0 and
len(examples) >= 10
}

result = validate_training_data("training_data.jsonl")
print(result)

```

Fine-Tuning with OpenAI

```

from openai import OpenAI
import time

client = OpenAI()

```



— 1. Upload training data

```
training_file = client.files.create(
    file=open("training_data.jsonl", "rb"),
    purpose="fine-tune"
)
validation_file = client.files.create(
    file=open("validation_data.jsonl", "rb"),
    purpose="fine-tune"
)
print(f"Training file ID: {training_file.id}")
```

— 2. Create fine-tuning job

```
job = client.fine_tuning.jobs.create(
    training_file = training_file.id,
    validation_file = validation_file.id,
    model = "gpt-4.1-mini-2025-04-14", # base
    model to fine-tune
    method = { # replaces the
        deprecated top-level `hyperparameters`
        "type": "supervised",
        "supervised": {
            "hyperparameters": {
                "n_epochs": 3, # 3-5 is typical;
                more = higher overfitting risk
                "batch_size": "auto",
                "learning_rate_multiplier": "auto",
            },
        },
    },
    suffix = "sql-generator" # appears in the model name:
    ft:gpt-4.1-mini-...:my-org:sql-generator:<id>
)
print(f"Job ID: {job.id}, status: {job.status}")
```

— 3. Monitor progress

```
while True:
    job = client.fine_tuning.jobs.retrieve(job.id)
    print(f"Status: {job.status}")
```

```

        if job.status in ("succeeded", "failed", "cancelled"):
            break
        # Check recent events
        for event in
client.fine_tuning.jobs.list_events(job.id, limit=5).data:
            print(f" [{event.created_at}] {event.message}")
        time.sleep(60)

print(f"Fine-tuned model: {job.fine_tuned_model}")
# ft:gpt-4.1-mini-2025-04-14:my-org:sql-generator:abc123

# — 4. Use the fine-tuned model

```

```

response = client.chat.completions.create(
    model=job.fine_tuned_model,
    messages=[
        {"role": "system", "content": "You are a SQL
expert. Generate ANSI SQL only."},
        {"role": "user", "content": "Show total revenue
by region for last quarter"}
    ]
)
print(response.choices[0].message.content)

```

Fine-Tuning with Hugging Face

For open-source models (Llama 3, Mistral, Gemma) on your own GPU or cloud VM.

```

pip install transformers datasets peft accelerate
bitsandbytes trl

```

```

from datasets import Dataset
from transformers import AutoTokenizer,
AutoModelForCausalLM, TrainingArguments
from peft import LoraConfig, get_peft_model
from trl import SFTConfig, SFTTrainer
import torch

MODEL_NAME = "meta-llama/Meta-Llama-3-8B-Instruct" #

```

gated: accept the licence on Hugging Face first; any causal LM works

— 1. Load model in 4-bit quantization (saves memory)

```
from transformers import BitsAndBytesConfig
```

```
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_use_double_quant=True,
    bnb_4bit_compute_dtype=torch.bfloat16,
)
```

```
tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
tokenizer.pad_token = tokenizer.eos_token
```

```
model = AutoModelForCausalLM.from_pretrained(
    MODEL_NAME,
    quantization_config=bnb_config,
    device_map="auto",
)
```

— 2. Apply LoRA

```
lora_config = LoraConfig(
    r=16, # rank – higher = more parameters,
    more capacity
    lora_alpha=32, # scaling factor (usually 2*r)
    target_modules=["q_proj", "v_proj", "k_proj",
    "o_proj"], # which layers to adapt
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM",
)
```

```
model = get_peft_model(model, lora_config)
model.print_trainable_parameters()
# Llama 3 8B, r=16, q/k/v/o_proj: trainable params:
13,631,488 (about 0.17% of the model).
# The 'all params' figure differs when the base model is
```

loaded in 4-bit.

— 3. Prepare dataset

Use the chat "messages" format: SFTTrainer then applies the model's own chat template, so training matches the prompt format used at inference. Hand-written special tokens (such as <|user|>) that the tokenizer does not define teach the model a format it never sees.

SYSTEM = "You are a SQL expert. Generate ANSI SQL only."

```
def to_messages(example):
    return {"messages": [
        {"role": "system", "content": SYSTEM},
        {"role": "user", "content":
example["question"]},
        {"role": "assistant", "content": example["sql"]},
    ]}

raw_data = [
    {"question": "Count orders by status", "sql": "SELECT
status, COUNT(*) FROM orders GROUP BY status;"},
    # ... more examples
]
dataset = Dataset.from_list(raw_data).map(to_messages,
remove_columns=["question", "sql"])
splits = dataset.train_test_split(test_size=0.1, seed=42)
# hold out data for eval
```

— 4. Train

SFTConfig extends TrainingArguments with SFT-specific options.

Tested with transformers 5.17, TRL 1.14 and PEFT 0.21; argument names change between major versions.

```
training_args = SFTConfig(
    output_dir = "./fine-tuned-model",
    num_train_epochs = 3,
    per_device_train_batch_size = 4,
    gradient_accumulation_steps = 4,
```

```

        warmup_steps          = 0.05,          # a float below 1 is a
ratio of total steps (transformers 4.x: warmup_ratio)
        learning_rate         = 2e-4,
        bf16                  = True,          # matches
bnb_4bit_compute_dtype; use fp16=True on GPUs without
bfloat16
        logging_steps         = 10,
        save_steps             = 100,
        eval_strategy          = "steps",      # was
evaluation_strategy in older transformers
        eval_steps             = 100,
        max_length             = 2048,        # was max_seq_length
in older TRL releases
    )

    trainer = SFTTrainer(
        model                  = model,
        args                   = training_args,
        train_dataset          = splits["train"],
        eval_dataset           = splits["test"],
        processing_class       = tokenizer,
    )

    trainer.train()
    trainer.save_model("./fine-tuned-model")

# — 5. Merge LoRA weights into base model for deployment

```

```

from peft import PeftModel

base_model =
AutoModelForCausalLM.from_pretrained(MODEL_NAME,
dtype=torch.float16) # torch_dtype in transformers 4.x
merged = PeftModel.from_pretrained(base_model, "./fine-
tuned-model")
merged = merged.merge_and_unload()
merged.save_pretrained("./merged-model")
tokenizer.save_pretrained("./merged-model")

```

LoRA / PEFT (Parameter-Efficient Fine-Tuning)

Why LoRA? Full fine-tuning keeps weights, gradients and optimizer state for every parameter: roughly 16 bytes per parameter with Adam in mixed precision, or about 110GB for a 7B model before activations. LoRA freezes the base weights and trains tiny added matrices, updating well under 1% of parameters, with quality that is comparable for many tasks at a fraction of the cost.

Full fine-tuning:

Original weights W (7B params) $\rightarrow$ updated W' (7B params)

GPU memory: ~16 bytes/param with Adam (~110GB for 7B)

Training time: days

LoRA:

Original weights W (frozen)

Two small matrices A ($r \times d$) and B ($d \times r$) where $r \ll d$

Update = $W + A \times B$ (only A and B are trained)

$r=16$ on the attention projections adds ~0.2% trainable parameters

GPU memory: roughly 8-16GB for a 7-8B model with 4-bit quantization (QLoRA)

Training time: hours

```
# LoRA hyperparameter guide
lora_config = LoraConfig(
    r=8,          # rank: 4-64; higher = more capacity, more
memory          # start with 8-16; increase if underfitting

    lora_alpha=16, # scaling: usually 1-2x rank; controls
magnitude of updates

    target_modules=["q_proj", "v_proj"], # which attention
layers to adapt
    # For most models: ["q_proj", "k_proj", "v_proj",
"o_proj"]
    # For aggressive adaptation add: ["gate_proj",
"up_proj", "down_proj"]

    lora_dropout=0.1, # regularization: 0.05-0.1 typical
)
```

Evaluating Fine-Tuned Models

```
import anthropic
import json

def evaluate_model(model_fn, test_cases: list[dict]) -> dict:
    """
    model_fn: callable(prompt) -> output string
    test_cases: [{"input": str, "expected": str, "check":
callable}]
    """
    results = []
    for case in test_cases:
        output = model_fn(case["input"])
        passed = case["check"](output, case["expected"])
        results.append({"input": case["input"], "output":
output, "passed": passed})

    pass_rate = sum(r["passed"] for r in results) /
len(results)
    return {"pass_rate": pass_rate, "details": results}

# Test cases for SQL generation
sql_test_cases = [
    {
        "input": "Count orders by status",
        "expected": "SELECT status, COUNT(*)",
        "check": lambda output, expected: expected.lower()
in output.lower()
    },
    {
        "input": "Top 5 customers by revenue",
        "expected": "LIMIT 5",
        "check": lambda output, expected: "LIMIT 5" in
output.upper() and "ORDER BY" in output.upper()
    },
]

# Always evaluate:
```

```
# 1. On a held-out test set (not in training data)
# 2. Against the base model (did fine-tuning actually help?)
# 3. On adversarial inputs (does it handle edge cases?)
# 4. For regressions (did it lose general capability?)
```

Dataset Construction Patterns

```
# Pattern 1: Generate synthetic data with a stronger model 
def generate_training_examples(task_description: str, n:
int = 100) -> list[dict]:
    """Use Claude to generate (input, output) pairs for
fine-tuning."""
    client = anthropic.Anthropic()
    response = client.messages.create(
        model="claude-sonnet-5-5",
        max_tokens=4096,
        messages=[{"role": "user", "content": f"""
Generate {n} diverse training examples for this fine-tuning
task:
{task_description}

Return as a JSON array: [{"input": "...", "output":
"..."}]

Requirements:
- Vary complexity from simple to complex
- Include edge cases
- Outputs must be exactly correct
- No explanations in outputs – output only
"""}]
    )
    return json.loads(next(b.text for b in response.content
if b.type == "text"))

# Pattern 2: Mine from existing system logs
def mine_from_logs(log_file: str) -> list[dict]:
    """Extract (user_query, good_response) pairs from
production logs."""
    examples = []
```

```

with open(log_file) as f:
    for line in f:
        entry = json.loads(line)
        if entry.get("feedback") == "thumbs_up": #
only use approved responses
            examples.append({
                "input": entry["user_message"],
                "output": entry["assistant_message"]
            })
    return examples

# Pattern 3: Human-curated corrections
# Store: original_output, corrected_output, corrected_by,
timestamp
# Fine-tune on: (input, corrected_output) pairs
# This is the highest quality data source

```

Production Considerations

```

# Serving a fine-tuned model
# Option 1: OpenAI fine-tuned model – just use the model ID
response = client.chat.completions.create(
    model="ft:gpt-4.1-mini-2025-04-14:my-org::abc123", #
returned as job.fine_tuned_model
    messages=[...]
)

# Option 2: Self-hosted with vLLM (open-source models)
# vllm serve ./merged-model --port 8000
# Then call like any OpenAI-compatible API:
from openai import OpenAI
local_client = OpenAI(base_url="http://localhost:8000/v1",
api_key="dummy")
response = local_client.chat.completions.create(
    model="./merged-model",
    messages=[...]
)

# Cost comparison – work it out for your own volume rather
than trusting rules of thumb:

```

```
# API model:    daily_tokens / 1e6 × price_per_million  
(input and output priced separately)  
#              e.g. 1M tokens/day at $2 per 1M ≈  
$2/day (look up current prices)  
# Fine-tuned:  training cost (one-off) + usually higher  
per-token inference price  
# Self-hosted: GPU instance hours (paid even when idle)  
+ engineering/ops time  
# Self-hosting only wins at high, steady volume or when  
data can't leave your network.
```

Common Pitfalls

1. Fine-tuning to add knowledge 
Problem: model still hallucinates – fine-tuning teaches behavior, not facts
Fix: Use RAG for factual grounding; use fine-tuning for style/format
2. Too little data
Problem: < 50 examples → model memorizes, doesn't generalize
Fix: Aim for 100-500 high-quality examples minimum; synthetic data helps
3. Low-quality training data
Problem: inconsistent outputs → model learns inconsistency
Fix: Have a human review every training example; quality >> quantity
4. Skipping evaluation
Problem: no way to know if fine-tuning helped or hurt
Fix: Build eval set before you start; measure base model vs fine-tuned
5. Over-training (too many epochs)
Problem: model memorizes training set, fails on new inputs
Fix: Monitor validation loss; stop when it stops

decreasing

6. No regression testing

Problem: fine-tuning improves SQL but breaks summarization

Fix: Eval on diverse capabilities, not just the target task

7. Using fine-tuning as a shortcut for better prompting

Problem: fine-tuning costs time and money

Fix: Exhaust prompt engineering first (few-shot, CoT, structured system prompt)

Cheat Sheet

Should you fine-tune? Try these first, in order

Step	Fixes	Cost
1. Better prompt + few-shot examples	Format, tone, simple domain rules	Minutes
2. Structured outputs / tool schemas	Format reliability	Minutes
3. RAG	Missing or changing knowledge	Days
4. A stronger model or higher effort	Reasoning quality	A config change
5. Fine-tuning	Consistent style/format at scale, a narrow task on a small cheap model, lower latency	Weeks: data, training, evals, hosting

Approach	What changes	Memory needed (7–8B model)	When
Full fine-tune	All weights	Very high (multiple large GPUs)	Large budgets; big domain shift
LoRA	Small adapter matrices	Moderate (one 24–48 GB GPU)	Default for open models

Approach	What changes	Memory needed (7–8B model)	When
QLoRA	LoRA on a 4-bit quantized base	Low (one 16–24 GB GPU)	Limited hardware
Hosted API fine-tuning	Provider-managed	None	Fastest path on supported models

LoRA knobs: r 8–64 (adapter capacity) · $\text{lo_ra_alpha} \approx 2 \times r$ · lo_ra_dropout 0.05–0.1 · target_modules attention projections (q_proj , v_proj , ...) · learning rate $\sim 1e-4$ to $2e-4$ · 1–3 epochs

Training data checklist: hundreds to a few thousand high-quality examples · the same prompt format you'll use at inference · deduplicated · no test examples leaked into training · a held-out eval split · PII removed · edge cases and refusals included

Chat-format JSONL record

```
{"messages": [{"role": "system", "content": "You write ANSI SQL."}, {"role": "user", "content": "Count orders by status"}, {"role": "assistant", "content": "SELECT status, COUNT(*) FROM orders GROUP BY status;"}]}
```

Interview Questions

Q: What's the difference between fine-tuning and prompt engineering? A: Prompt engineering changes the input without modifying the model — fast, cheap, reversible. Fine-tuning changes the model's weights by training on examples — more powerful for consistent behavior and style, but requires data, compute, and eval infrastructure. Start with prompts; only fine-tune when prompts can't achieve the goal consistently.

Q: What is LoRA and why is it preferred over full fine-tuning? A: LoRA (Low-Rank Adaptation) freezes the pre-trained model weights and trains two small low-rank matrices that are added to the original weights. It updates well under 1% of parameters instead of 100%, cutting GPU memory from roughly 110GB to about 8–16GB for a 7–8B model when combined with 4-bit quantization (QLoRA), and training time from days to hours, with comparable quality on many tasks.

Q: When would you choose fine-tuning over RAG? A: RAG is better for knowledge (facts that change, need citations). Fine-tuning is better for behavior (consistent format, style, custom classifications, domain-specific extraction). Often the right answer is both: fine-tune the model for behavior, add RAG for knowledge grounding.

Further Reading

- [Hugging Face TRL — SFTTrainer](https://huggingface.co/docs/trl/sft_trainer) (https://huggingface.co/docs/trl/sft_trainer)
- [Hugging Face PEFT](https://huggingface.co/docs/peft/index) (https://huggingface.co/docs/peft/index) — LoRA and other adapter methods
- [OpenAI fine-tuning guide](https://developers.openai.com/api/docs/guides/model-optimization) (https://developers.openai.com/api/docs/guides/model-optimization)
- [Unsloth](https://docs.unsloth.ai/) (https://docs.unsloth.ai/) — faster, lower-memory LoRA/QLoRA fine-tuning
- *LoRA: Low-Rank Adaptation of Large Language Models* — Hu et al., 2021 · *QLoRA* — Dettmers et al., 2023
- [Eval & Evals](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/) — measure the fine-tuned model against your baseline before shipping

Previous: [Claude Code](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/claude-code/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/claude-code/) · **Next:** [AI Observability](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-observability/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-observability/) · **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (https://sarangambekar1997.github.io/data-engineering-handbook/)

 September 28, 2026