When to Fine-Tune vs Alternatives

Fine-tuning is expensive and slow compared to prompting or RAG -- but when you genuinely need to change model behavior, style, or domain expertise, nothing else works. Use this table to avoid fine-tuning when a simpler approach will do.

Approach	Best When	Cost	Effort	Latency Impact
Prompt engineering	Task is well-defined, few formats	Very low	Low	None
Few-shot examples	Need consistent format/style	Low	Low	Slightly higher (more tokens)
RAG	Need up-to-date or private knowledge	Medium	Medium	Higher (retrieval step)
Fine-tuning	Need behavior change, style, or domain expertise	High	High	Lower at inference
Pre-training (continued)	Entirely new domain/language	Very high	Very high	None

Decision Tree

```
Can prompt engineering solve it?
  YES -> Stop, use prompts
  NO  -> Is the issue missing knowledge?
           YES -> Use RAG
           NO  -> Is it about style, format, or behavior?
                    YES -> Fine-tune
                    NO  -> Is the base model too slow/expensive?
                             YES -> Fine-tune smaller model
                             NO  -> Revisit prompt engineering
```

Data Preparation

Data quality is the single biggest predictor of fine-tuning success -- a model trained on 200 clean, diverse examples will outperform one trained on 2,000 noisy ones. Get the format and quality right before you spend a dollar on compute.

JSONL Format (OpenAI Chat)

```
{"messages": [{"role": "system", "content": "You are a legal assistant."}, {"role": "user", "conte
{"messages": [{"role": "system", "content": "You are a legal assistant."}, {"role": "user", "conte
```

JSONL Format (Anthropic)

```
{"messages": [{"role": "user", "content": "Summarize this clause..."}, {"role": "assistant", "cont
```

JSONL Format (Alpaca-style, for open models)

```
{"instruction": "Summarize the following clause", "input": "The party of the first part...", "outp
```

Data Quality Checklist

Check	Why	How
Minimum 50-100 examples	Models need enough signal	Count rows
Consistent format	Reduces noise	Validate JSON schema
Diverse inputs	Prevents overfitting	Cluster by topic, check distribution
Clean outputs	Garbage in = garbage out	Human review sample
No contradictions	Confuses the model	Cross-check examples
Balanced classes	Prevents bias toward majority	Check class distribution
Deduplication	Overfitting on duplicates	Hash-based dedup
PII removed	Privacy compliance	Run PII scanner

Recommended Dataset Sizes

Task Type	Minimum	Good	Excellent
Classification	50	500	5,000+
Structured extraction	100	1,000	10,000+
Style/tone transfer	200	1,000	5,000+
Domain expertise	500	5,000	50,000+
Code generation	1,000	10,000	100,000+

Fine-Tuning Methods

Full fine-tuning is rarely necessary -- LoRA and QLoRA achieve 90-95% of full fine-tuning quality at a fraction of the memory and cost. Choose your method based on your GPU budget and quality requirements.

Method	Parameters Trained	Memory Required	Training Speed	Quality
Full fine-tuning	All	Very high (4x model)	Slow	Highest
LoRA	0.1-1% of params	Low (model + adapters)	Fast	High
QLoRA	0.1-1% of params	Very low (4-bit base)	Fast	High

Method	Parameters Trained	Memory Required	Training Speed	Quality
Prefix tuning	Prefix embeddings only	Very low	Very fast	Medium
Prompt tuning	Soft prompt tokens	Minimal	Fastest	Lower
RLHF	Reward model + policy	Very high	Very slow	Highest for alignment
DPO	Policy only (no reward model)	High	Medium	High for alignment

LoRA Key Concepts

```
Original weight matrix: W (d x d)
LoRA decomposition:    W' = W + BA
  where B: (d x r), A: (r x d), r << d

- r (rank): Typically 4-64. Higher = more capacity, more memory
- alpha: Scaling factor, typically 2x rank
- Target modules: q_proj, v_proj (minimum), k_proj, o_proj, up_proj, down_proj (full)
```

LoRA Configuration Example

```
from peft import LoraConfig, get_peft_model

config = LoraConfig(
    r=16,                          # rank
    lora_alpha=32,                 # scaling (usually 2*r)
    target_modules=[               # which layers to adapt
        "q_proj", "v_proj",
        "k_proj", "o_proj",
        "up_proj", "down_proj"
    ],
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM"
)
model = get_peft_model(base_model, config)
model.print_trainable_parameters()
# trainable: ~0.5% of total parameters
```

QLoRA Setup

```
from transformers import BitsAndBytesConfig

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

```
model = AutoModelForCausalLM.from_pretrained(
    "meta-llama/Llama-3.1-8B",
    quantization_config=bnb_config,
    device_map="auto"
)
# Then apply LoRA on top of 4-bit model
```

Training Parameters

Wrong hyperparameters waste compute and produce bad models. Learning rate and epoch count are the two most impactful settings -- get those right first, then tune the rest.

Parameter	Typical Range	Notes
Learning rate	1e-5 to 5e-4	Lower for larger models
Batch size	4-32	Limited by GPU memory; use gradient accumulation
Epochs	1-5	More data = fewer epochs needed
Warmup steps	5-10% of total steps	Stabilizes early training
Weight decay	0.01-0.1	Regularization
Max sequence length	512-4096	Match your data; padding wastes compute
Gradient accumulation	2-16	Simulates larger batch size
LR scheduler	Cosine or linear decay	Cosine usually better

Training with Hugging Face

```
from transformers import TrainingArguments, Trainer
from trl import SFTTrainer

training_args = TrainingArguments(
    output_dir="./output",
    num_train_epochs=3,
    per_device_train_batch_size=4,
    gradient_accumulation_steps=4,
    learning_rate=2e-4,
    warmup_ratio=0.05,
    lr_scheduler_type="cosine",
    weight_decay=0.01,
    logging_steps=10,
    save_strategy="epoch",
    evaluation_strategy="epoch",
    bf16=True,
    gradient_checkpointing=True,
)

trainer = SFTTrainer(
    model=model,
    train_dataset=train_data,
```

```
    eval_dataset=eval_data,
    args=training_args,
    max_seq_length=2048,
)
trainer.train()
```

GPU Memory Estimation

Running out of GPU memory mid-training is the most common fine-tuning blocker. Use this table to right-size your hardware before you start, and pick QLoRA when your GPU budget is tight.

Model Size	Full FT	LoRA (r=16)	QLoRA (4-bit)
1B	~8 GB	~5 GB	~3 GB
7B	~56 GB	~18 GB	~8 GB
13B	~104 GB	~32 GB	~14 GB
70B	~560 GB	~160 GB	~48 GB

Rule of thumb: Full fine-tuning needs ~4x model size in memory (model + gradients + optimizer states).

OpenAI Fine-Tuning (API)

If you do not need to self-host, API-based fine-tuning is the fastest path to a custom model. You upload data, kick off a job, and get a model endpoint back -- no GPU management required.

```
# 1. Upload training file
file = client.files.create(
    file=open("training.jsonl", "rb"),
    purpose="fine-tune"
)

# 2. Create fine-tuning job
job = client.fine_tuning.jobs.create(
    training_file=file.id,
    model="gpt-4o-mini-2024-07-18",
    hyperparameters={
        "n_epochs": 3,
        "batch_size": "auto",
        "learning_rate_multiplier": "auto"
    }
)

# 3. Check status
client.fine_tuning.jobs.retrieve(job.id)

# 4. Use fine-tuned model
response = client.chat.completions.create(
    model="ft:gpt-4o-mini-2024-07-18:org:name:id",
    messages=[...]
)
```

Evaluation

A fine-tuned model that looks great on training loss can be catastrophically overfit. Always evaluate on held-out data using task-specific metrics, and compare against the base model to confirm the fine-tuning actually helped.

Metric	What It Measures	When to Use
Loss (train/val)	Model fitting	Always (check for overfitting)
Exact match	Perfect output match	Classification, extraction
BLEU / ROUGE	Text overlap	Summarization, translation
BERTScore	Semantic similarity	Open-ended generation
Human eval (side-by-side)	Overall quality	Before production deployment
Task-specific accuracy	Domain performance	Always

Overfitting Signals

Signal	What It Looks Like	Fix
Train loss drops, val loss rises	Classic overfit curve	Reduce epochs, add data
Val loss flat, train loss dropping	Memorization starting	Early stopping
Perfect train accuracy	Model memorized data	More data, regularization
Worse than base model on general tasks	Catastrophic forgetting	Lower LR, fewer epochs, LoRA

Cost Comparison

Fine-tuning costs vary by orders of magnitude depending on whether you use an API provider or self-host. Factor in both training cost and ongoing inference cost when making your decision.

Provider	Model	Training Cost	Inference Cost
OpenAI	GPT-4o-mini FT	$3.00/1M train tokens	$0.60/$2.40 per 1M in/out
OpenAI	GPT-4o FT	$25.00/1M train tokens	$3.75/$15.00 per 1M in/out
Together.ai	Llama 8B	~$0.50/hr (1x A100)	$0.20/1M tokens
Self-hosted	Any open model	GPU cost only	GPU cost only

Common Pitfalls

The most expensive mistake is fine-tuning when you should have used RAG, and the second most expensive is overfitting because you skipped the eval set. This list covers both and everything in between.

Pitfall	Problem	Fix
Fine-tuning for knowledge	LLMs forget; hallucination risk	Use RAG for knowledge, fine-tune for behavior
Too few examples	Underfitting, poor generalization	Start with 200+ high-quality examples
Too many epochs	Overfitting	Monitor val loss, use early stopping
Inconsistent data format	Confuses the model	Standardize all examples
No eval set	Can't detect overfitting	Hold out 10-20% of data
Ignoring base model capabilities	Redundant training	Test base model + prompt first
Not merging LoRA for production	Extra inference overhead	Merge adapters into base weights
Training on generated data only	Model amplifies errors	Include human-validated examples