Gen AI: LLM Capabilities and Alignment

The Stages of Building LLM Apps

The Stages of Building Large Language Models

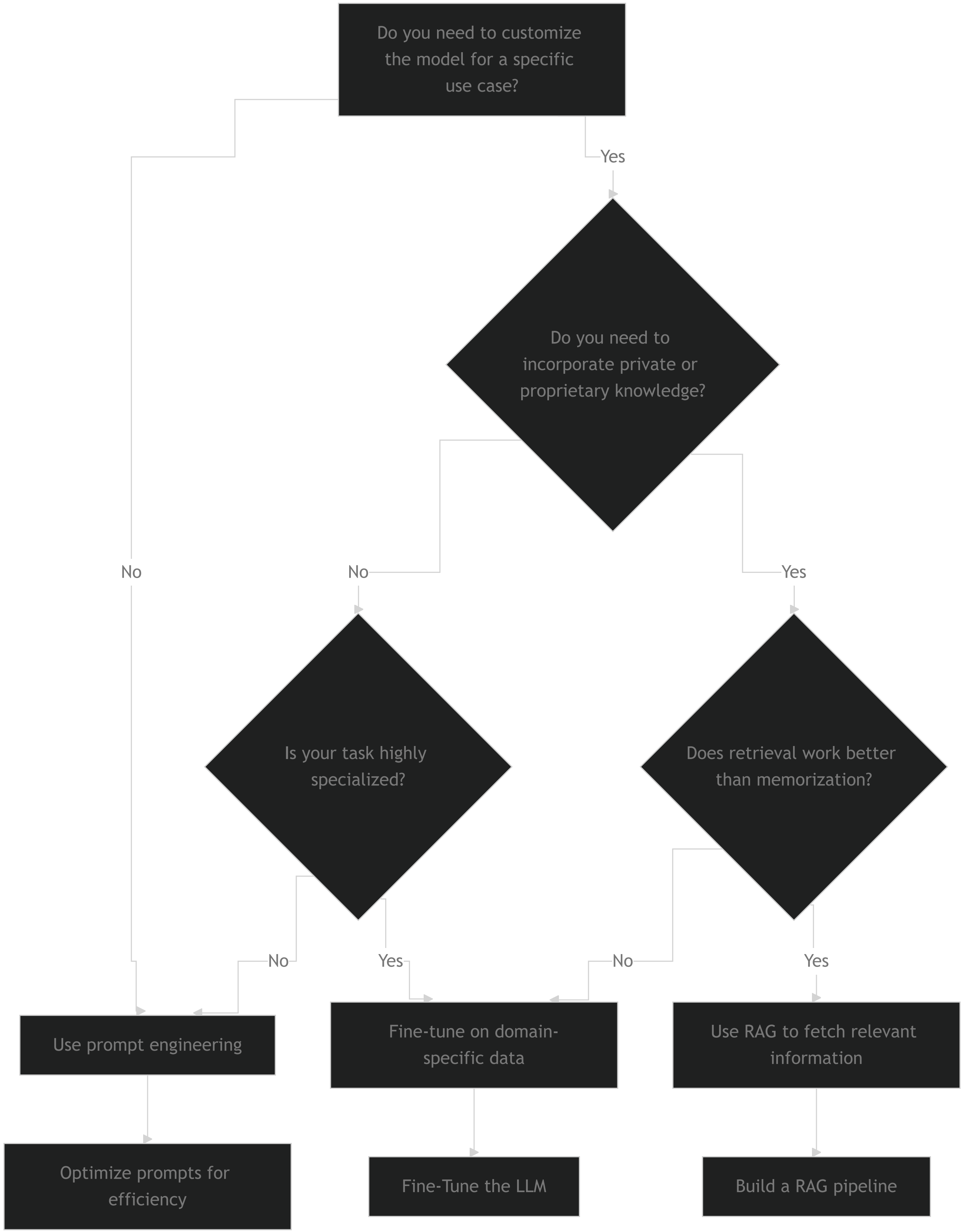
Learning Objective

By the end of this lesson, you will be able to:

- **Explain** the steps involved in building, fine-tuning, and deploying LLMs
- **Assess** situations to decide when fine-tuning is necessary
- **Describe** how pretrained models and multimodal capabilities fit into different use cases, and how to address optimization challenges.

When Do You Need to Fine-Tune an LLM?

Many organizations assume that fine-tuning is always required, but that's not the case. Below is a **decision framework** for choosing between **fine-tuning, retrieval-augmented generation (RAG), or prompt engineering**:


Do you need to customize the model for a specific use case?
Yes
No
Do you need to incorporate private or proprietary knowledge?
No
Yes
Is your task highly specialized?
Does retrieval work better than memorization?
No
Yes
No
Yes
Use prompt engineering
Fine-tune on domain-specific data
Use RAG to fetch relevant information
Optimize prompts for efficiency
Fine-Tune the LLM
Build a RAG pipeline

Hands-On Walkthrough: Fine-Tuning a Small LLM

Goal: Fine-tune a **DistilBERT** model on a synthetic dataset and compare its performance to a base model with prompt engineering.

Step 1: Install Dependencies

Copy

```
pip install transformers datasets torch
```

Step 2: Load a Pretrained Model and Dataset

Copy

```
from transformers import AutoModelForSequenceClassification, AutoTokenizer, Train
from datasets import load_dataset

# Load dataset and model
dataset = load_dataset("imdb")  # Example dataset
tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")
model = AutoModelForSequenceClassification.from_pretrained("distilbert-base-uncas
```

Step 3: Preprocess the Data

Copy

```
def tokenize_function(examples):
    return tokenizer(examples["text"], padding="max_length", truncation=True)

tokenized_datasets = dataset.map(tokenize_function, batched=True)
train_dataset = tokenized_datasets["train"].shuffle(seed=42).select(range(1000))
```

Step 4: Fine-Tune the Model

Copy

```
training_args = TrainingArguments(output_dir="./results", evaluation_strategy="ep

trainer = Trainer(model=model, args=training_args, train_dataset=train_dataset)
trainer.train()
```

Step 5: Compare Fine-Tuned Model to Prompt Engineering

Copy

```
inputs = tokenizer("This movie was absolutely fantastic!", return_tensors="pt")
outputs = model(**inputs)
print(outputs.logits)
```

- **Reflection**: Does fine-tuning **significantly outperform** a base model with well-crafted prompts?

Hands-On Optimization Task: Debugging Training Inefficiencies

Now, let’s optimize a training process by fixing inefficient code. **Identify and fix the inefficiencies in the code below.**

Copy

```
# Inefficient training process
for epoch in range(5):  # Too many epochs for small datasets
    for batch in train_dataset:
        inputs = tokenizer(batch["text"], return_tensors="pt")  # Repeated tokeni
        outputs = model(**inputs)
        loss = outputs.loss
        loss.backward()
        optimizer.step()
```

▶ 🤔 **What’s wrong?**

Optimized Version

Copy

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(train_dataset, batch_size=8, shuffle=True)

for epoch in range(2):  # Reduced epochs
    for batch in train_dataloader:
        inputs = tokenizer(batch["text"], return_tensors="pt", padding=True, trun
        outputs = model(**inputs)
        loss = outputs.loss
        loss.backward()
        optimizer.step()
```

Exploring Multimodal LLMs

Why Fine-Tuning Isn't Always Enough

- Some tasks require **more than just text** (e.g., images, audio, video).
- **Fine-tuning on text alone won't improve multimodal understanding.**
- Instead, use **embeddings** to combine modalities.

Quick Code Demo: Image + Text Model

Copy

```
from transformers import CLIPProcessor, CLIPModel
from PIL import Image

model = CLIPModel.from_pretrained("openai/clip-vit-base-patch32")
processor = CLIPProcessor.from_pretrained("openai/clip-vit-base-patch32")
image = Image.open("example.jpg")
inputs = processor(text=["a cat", "a dog"], images=image, return_tensors="pt", pa
```

```
outputs = model(**inputs)
print(outputs.logits_per_image)  # Image-text similarity scores
```

Bridging to RAG (Retrieval-Augmented Generation)

Now that we've explored **fine-tuning and optimization**, let's consider an alternative approach:

> **If fine-tuning doesn't work well for factual consistency, how do we improve LLMs?**

- Fine-tuning is **bad at retrieving updated facts**.
- Instead of making an LLM memorize facts, **RAG allows real-time information retrieval**.

< Previous

Next >