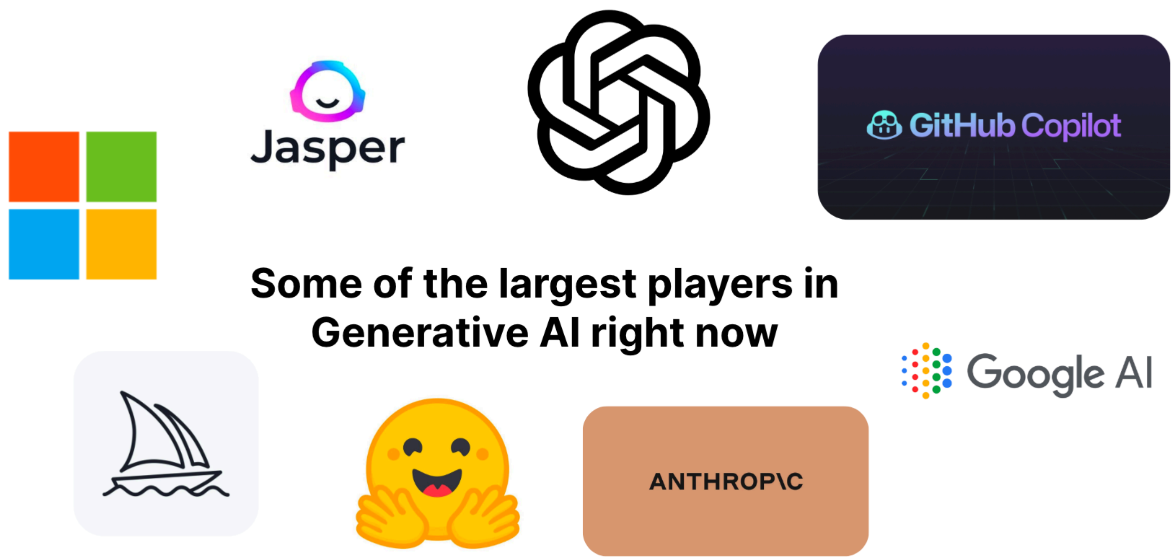
# Generative AI for Data Science

## Structure

1. What is Generative AI?
2. How can you use it?
3. Going Beyond ChatGPT: API & Functions
4. Langchain & Beyond: Using LLMs in Applications
5. Shortcomings
6. Further Reading

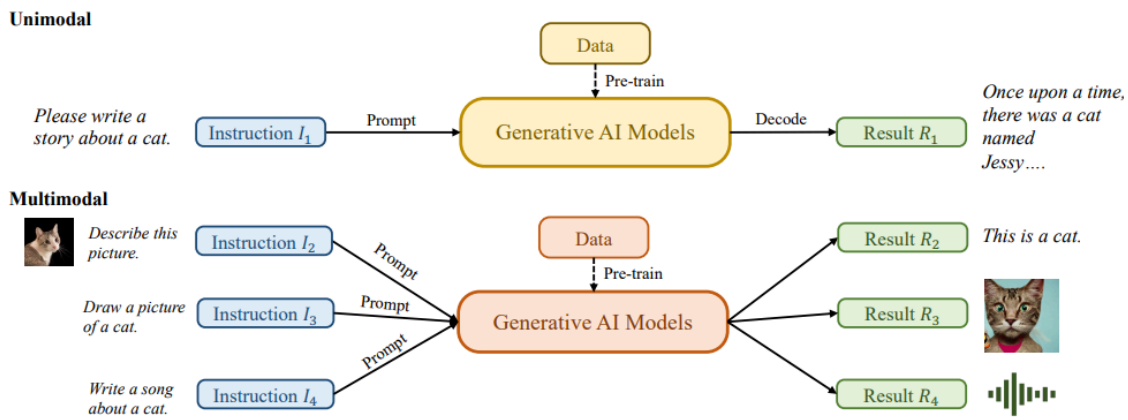
The notebook to replicate this lecture's tutorials can be found in today's "Challenge" on Kitt.

# 1. What is Generative AI?




## Gen AI is an umbrella term

- Model is trained
- (Optional) Model fine-tuned
- Inference is run
- Images, text, sound

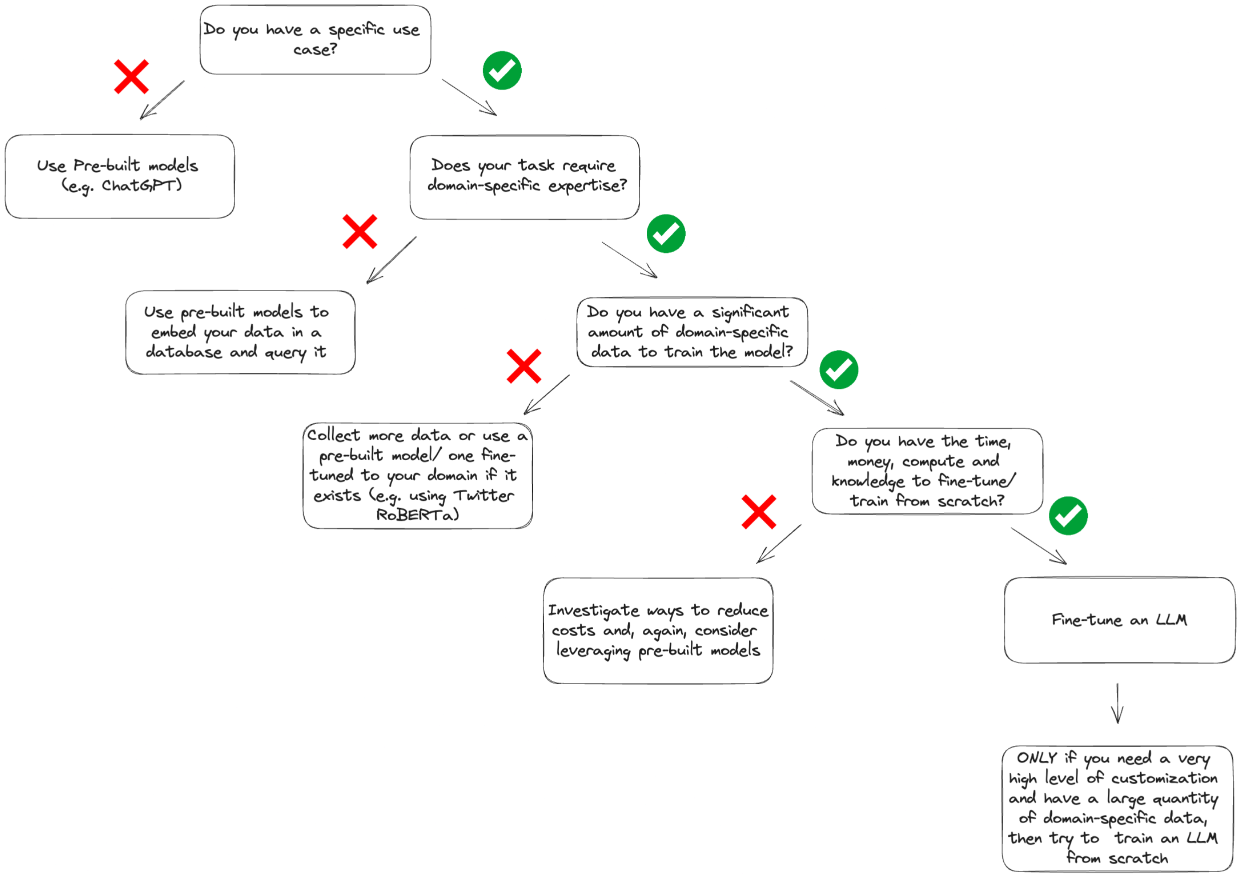



## 2. How can you use it?

The simplest (and most widely known) way to interact with high-quality generative AI is through ChatGPT:

- Trained on a vast amount of data
- 175+ billion trained parameters
- 700,000 dollars inference/ day (on top of 2-5 million dollars estimated cost for each training)

## Pre-trained vs fine-tuning vs from-scratch




## Prompt engineering:

Some key points:

- Using role-playing
- Being specific in the task
- Highlighting inputs and specifying outputs
- "Zero-shot" vs "few-shot"
- Using Chain-of-thought prompting

# 3. Going beyond ChatGPT

## The OpenAI API

```
import openai

# Set your OpenAI API key
openai_api_key = 'api-key-here'

# Initialize the OpenAI API client
openai.api_key = openai_api_key
```

```python
# Prompt for the AI model
prompt = "Translate the following English text to French: 'Hello, how are you?'"

# Make a request to the API to generate text
response = openai.chat.completions.create(
  model="gpt-3.5-turbo",  # Use the engine of your choice
  messages = [{"role": "user", "content": prompt}],
  max_tokens = 50
)
```

```python
response.choices[0].message.content
```

## System prompts

```python
# Prompt for the AI model
prompt = "Give instructions to cook vegetable samosas"

# Make a request to the API to generate text
response = openai.chat.completions.create(
  model="gpt-3.5-turbo",  # Use the engine of your choice
  messages = [{"role": "system", "content": "You are a sassy culinary instructor that gives sarcastic replies"},
              {"role": "user", "content": prompt}],
  max_tokens = 50
)
```

```python
response.choices[0].message.content
```

## Function calling: Imagine a function we might write

```python
def get_current_weather(location, unit):
  ### A request is made to an API with a specific format
  ### returns some result
```

```python
completion = openai.chat.completions.create(
  model="gpt-4",
  messages=[{"role": "user", "content": "I'm interested in the weather in Bozeman. I'm old-school so like it in F?"}],
  functions=[
  {
    "name": "get_current_weather",
    "description": "Get the current weather in a given location",
    "parameters": {
```

```
            "type": "object",
            "properties": {
                "location": {
                    "type": "string",
                    "description": "The city with its accompanying state, e.g. San Francisco, CA",
                },
                "unit": {"type": "string",
                         "enum": ["celsius", "fahrenheit"]},
            },
            "required": ["location"],
        },
    }
],
function_call="auto",
)
```

```
completion.choices[0].message.function_call.arguments
```

## A practical example

```
import pandas as pd
import json

df =
pd.read_csv("https://wagon-public-datasets.s3.amazonaws.com/deep_learning_datasets/results.csv"
)

df["date"] = pd.to_datetime(df["date"])
```

```
completion = openai.chat.completions.create(
    model="gpt-4",
    messages=[{"role": "user", "content": "Tell me about matches that took place in Italy between 1980
up until the end of the 20th century"}],
    functions=[
    {
        "name": "get_matches",
        "description": "Return the rows in a DataFrame about women's football games which satisfy the
criteria",
        "parameters": {
            "type": "object",
            "properties": {
                "country": {
                    "type": "string",
                    "description": "The name of the country the matches took place e.g. France or China",
                },
```

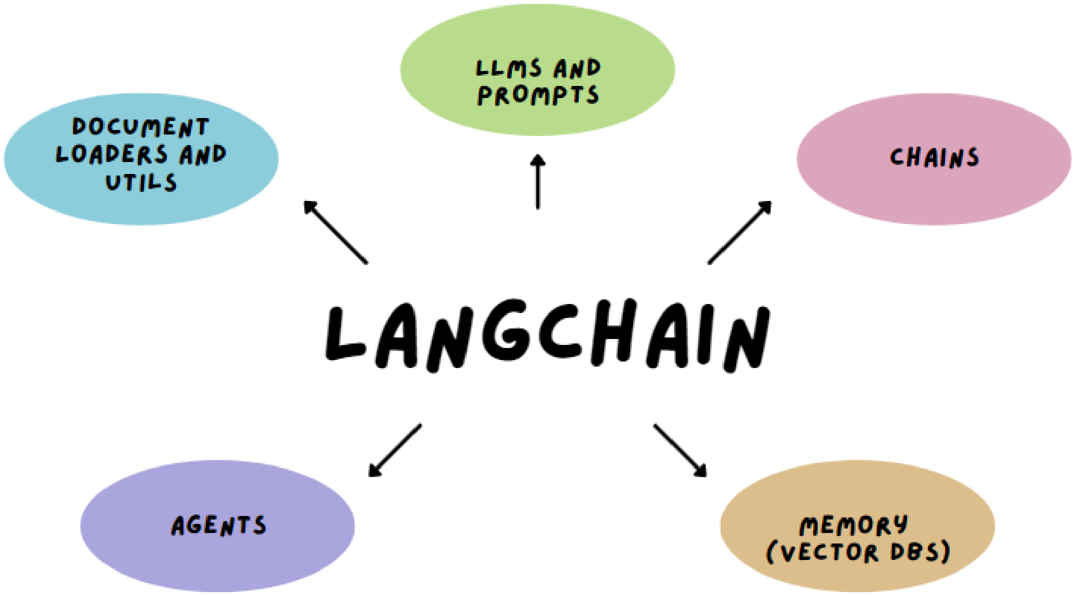
```
                "start_year": {
                    "type": "number",
                    "description": "The year to begin filtering from e.g. 1956",
                },
                "end_year": {
                    "type": "number",
                    "description": "The year to end filtering on e.g. 2005"}
            },
            "required": ["location", "start_year", "end_year"],
        },
    }
],
function_call="auto",
)
```

```
args = json.loads(completion.choices[0].message.function_call.arguments)

print(args)
```

```
def matches_finder(country: str, start_year: int, end_year: int):
    return df.loc[
        (df["country"] == country) &
        (start_year <= df["date"].dt.year) &
        (df["date"].dt.year <= end_year)
    ]
```

```
matches_finder(**args)
```

## 4. Langchain and Beyond:




## How can I work with larger amounts of data?

- We saw in the Transformers lecture how tricky it is to have large context windows (a.k.a. sequence length)
- ChatGPT and other models have ~32k tokens max

Does that mean that we can only ever work with documents <32k tokens 🥺?
We can use a Vector DataBase to store our embeddings 💪

We can use services like Open AI's [embeddings API](embeddings API) to convert large documents into vector representations and then store them 💪

```
model = "text-embedding-ada-002"

embedding = openai.embeddings.create(input=["""This is a simple embedding of a sentence"""],
                                     model=model)
```

How large are the embeddings we got?

```
import numpy as np
```

```
np.array(embedding["data"][0]["embedding"]).shape
```

## How can we go about tackling larger documents?

```
! wget -O book.pdf "https://greenteapress.com/thinkpython2/thinkpython2.pdf"
```

```
from langchain.document_loaders import PyPDFLoader

loader = PyPDFLoader("book.pdf")

data = loader.load()
```

```
print (f'You have {len(data)} documents in your data')
print (f"There are ~{np.mean([len(x.page_content) for x in data])} characters per document")
```

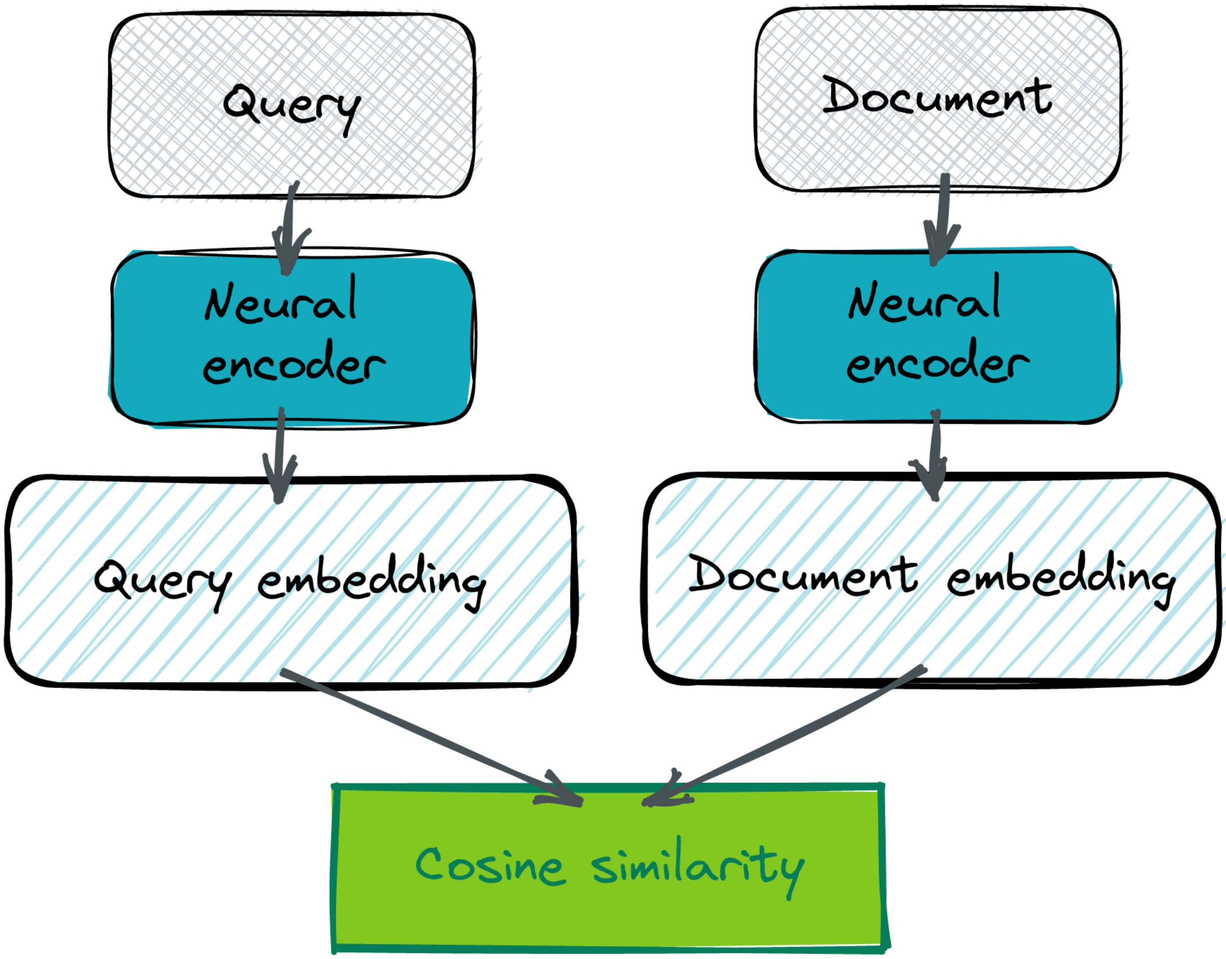
## How could we split our documents up?

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(chunk_size=2000, chunk_overlap=400)

texts = text_splitter.split_documents(data)
```

## Storing them in a Vector DataBase

```
from langchain.vectorstores import Chroma
from langchain_openai import OpenAIEmbeddings

embeddings = OpenAIEmbeddings(openai_api_key=api_key)
```

```
vector_db = Chroma.from_documents(texts, embeddings)
```




```
query = "How do I establish a Class?"
docs = vector_db.similarity_search(query, k = 5)
```

### Can we go even further?

```
from langchain_openai import ChatOpenAI
from langchain.chains.question_answering import load_qa_chain
```

```
llm = ChatOpenAI(temperature=0, openai_api_key=api_key)
chain = load_qa_chain(llm, chain_type="map_reduce")
```

🔎 A note on [temperature]() and on ["map_reduce"]()!

```
query = "How does the author recommend I keep studying after the book?"
docs = vector_db.similarity_search(query, k=1)
```

```
chain.run(input_documents=docs, question=query)
```

# Running LLMs locally/ privately

Why might you need to do this?

- Data privacy
- Fine-tuning on specific datasets

We can even download quantized (reduced) versions of very large models from HuggingFace 😮

## Why Quantize?

Assuming weights are stored in 32-bit float format:

1 model parameter = 4 bytes

1 billion parameters = 4 x 1,000,000,000 bytes = 4 GB (not even counting optimizer, gradient and activation info)

Many cutting edge models (Falcon, Llama, GPT 4) easily break 70 billion trainable parameters 🤯

```
output = llm("Q: How large is the earth's diameter? A: ",
         max_tokens=200,
         echo=True)
output["choices"][0]["text"]
```

## Running multi-modal models yourself (Colab recommended)

```
from diffusers import AutoPipelineForText2Image
import torch

pipeline = AutoPipelineForText2Image.from_pretrained(
    "runwayml/stable-diffusion-v1-5",
    torch_dtype=torch.float16,
    use_safetensors=True
).to("cuda") # For use w/ a GPU in colab
```

```
prompt = "A Renaissance painting of the Eiffel tower"
pipeline(prompt, num_inference_steps=30).images[0]
```

# 5. Shortcomings

- Bias in the model
- Reliance on LLMs for labelling
- Reliability (even with the Functions API)
- Recency of data
- Confidence intervals (or lack thereof)
- More in Ethics & AI!

# 6. Further Reading

- OpenAI API Docs: Filled with code examples to use
- Andrew Ng's Prompt Engineering for Developers: Excellent, free 1-hour course
- Full list of Deeplearning.ai courses: Build on many of the use cases mentioned in this lecture
- RSS Data Science and AI Newsletter: Monthly updates on latest tools
- HuggingFace Blog Post on QLora