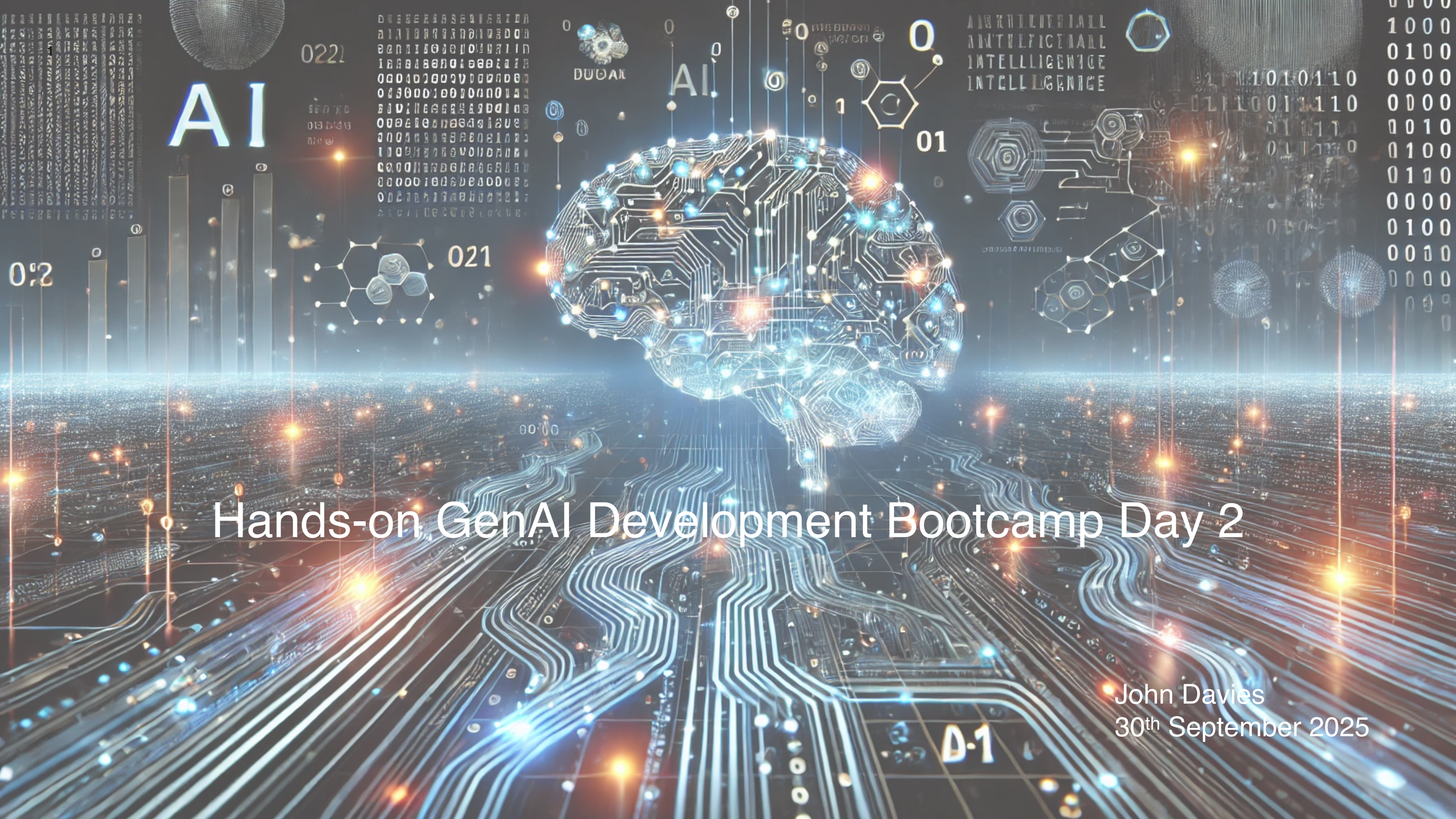


Hands-on GenAI Development Bootcamp Day 2

John Davies
30th September 2025

AI Astrology

```python
import ollama
from datetime import date
from ollama import Options
from rich.console import Console

def main():
    today = date.today().strftime('%A, %d-%m-%Y')
    LLM = "qwen3"

    name = input("Enter your name: ")
    star_sign = input("Enter your star sign: ")

    system_prompt = f"""You are an AI astrology assistant called Maude. Provide a short but interesting, positive and
        optimistic horoscope for tomorrow. Provide the response in Markdown format.
        Remember, the user is looking for a positive and optimistic outlook on their future.
        Use British English, metric and EU date formats where applicable."""

    instruction = f"Please provide a horoscope for {name} who's star sign is {star_sign}. Today's date is {today}."

    response = ollama.chat( model=LLM, think=True, stream=False,
        messages=[ {'role': 'system', 'content': system_prompt}, {'role': 'user', 'content': instruction} ],
        options=Options( temperature=0.8, num_ctx=4096, top_p=0.95, top_k=40, num_predict=-1 ))

    console = Console()

    if hasattr(response.message, 'thinking') and response.message.thinking:
        console.print(f"[bold blue]🤔 Maude's Thinking Process:[/bold blue]\n[dim]{response.message.thinking}[/dim]")
        console.print("\n" + "=" * 50 + "\n")

    console.print("[bold magenta]✨ Your Horoscope:[/bold magenta]")
    console.print(response.message.content)

if __name__ == "__main__":
    main()
```

AI Astrology

```python
import os
from groq import Groq
from datetime import date
from rich.console import Console
from rich.markdown import Markdown

client = Groq( api_key=os.environ.get("GROQ_API_KEY"))

def main():
    today = date.today().strftime('%A, %d-%m-%Y')
    LLM = "qwen/qwen3-32b"

    name = input("Enter your name: ")
    star_sign = input("Enter your star sign: ")

    system_prompt = f"""You are an AI astrology assistant called Maude. Provide a short but interesting, positive and
        optimistic horoscope for tomorrow. Provide the response in Markdown format.
        Remember, the user is looking for a positive and optimistic outlook on their future.
        Use British English, metric and EU date formats where applicable."""
    instruction = f"Please provide a horoscope for {name} who's star sign is {star_sign}. Today's date is {today}."

    response = client.chat.completions.create(
        reasoning_effort="default", stream=False, model=LLM,
        messages=[{'role': 'system', 'content': system_prompt}, {'role': 'user', 'content': instruction, }],
    )
    console = Console()

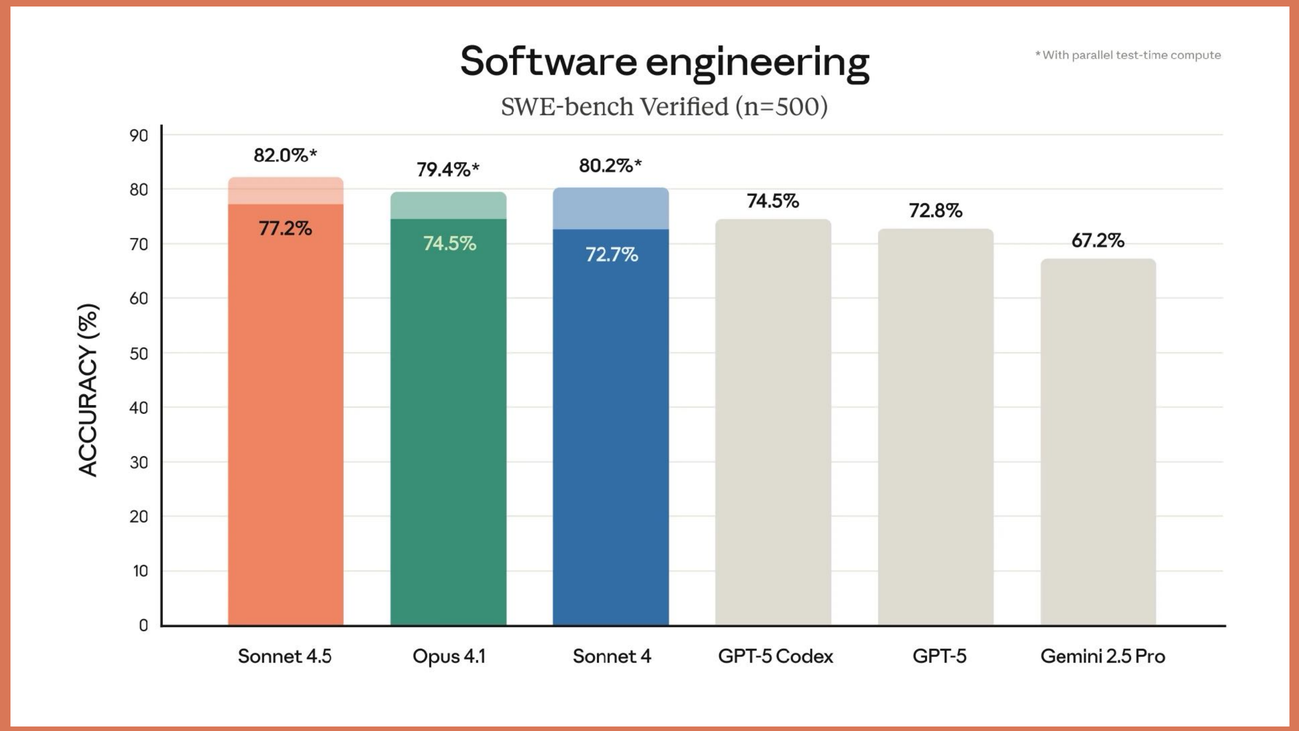
    # Render the Markdown content
    markdown = Markdown(response.choices[0].message.content)
    console.print(markdown)

if __name__ == "__main__":
    main()
```

Breaking News!

Claude Sonnet 4.5

- “the best coding model in the world”



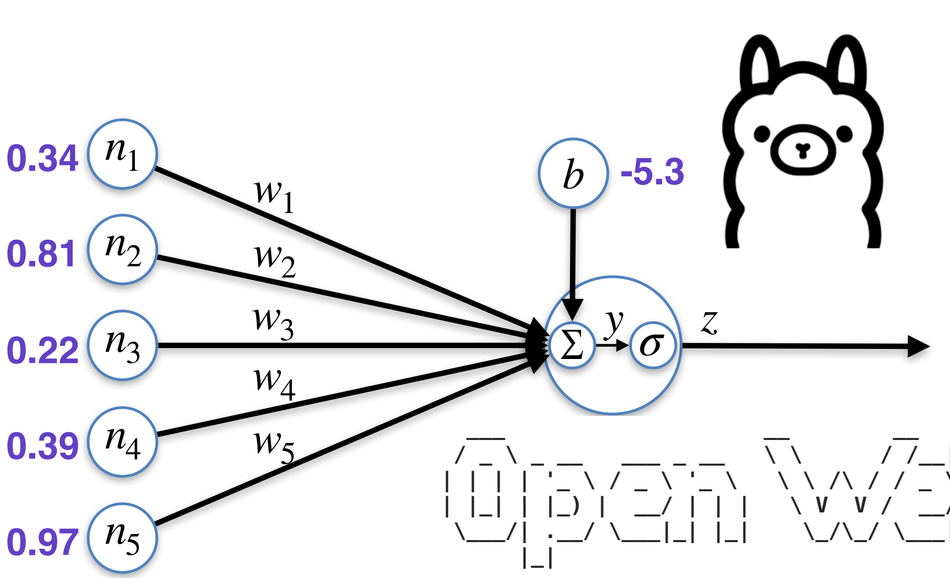
A quick recap of yesterday

We had a quick intro to Python & Ollama

- Keep an eye on Ollama version and new models
- You've also looked at other tools such as Open-WebUI

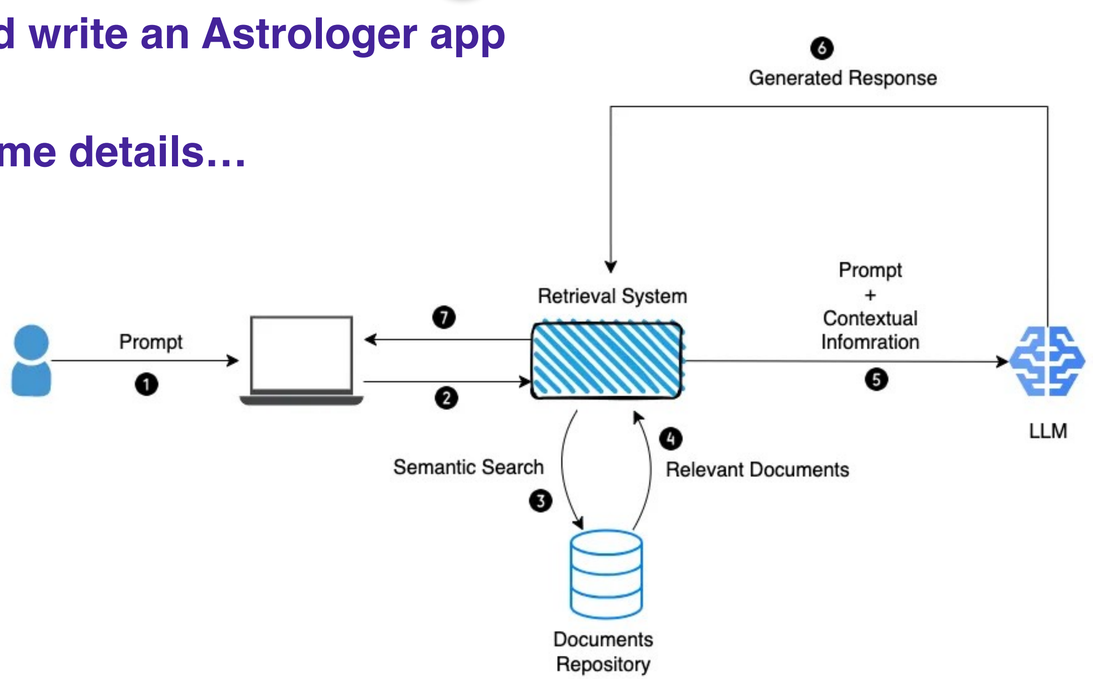
We covered Machine Learning and Neural Networks

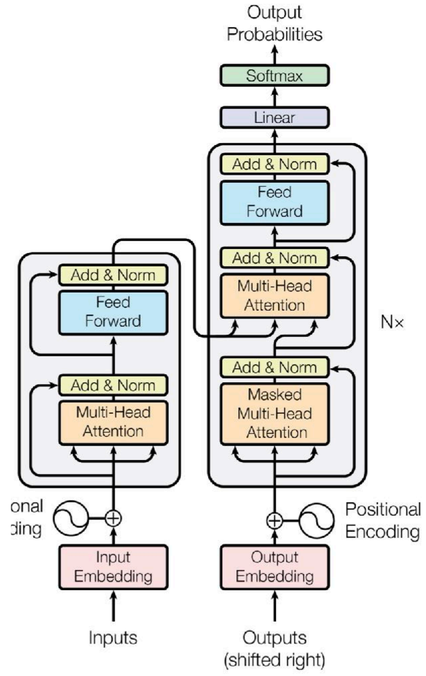
We went into some detail on the LLM and attention heads



We had some time to play with the LLMs and write an Astrologer app

We got a good overview of RAG, now for some details…





Chain-of-Thought (CoT)

CoT is the process of breaking down the task into smaller tasks and then following those, one by one

- This seems like common sense and it effectively is exactly that, humans do this too
- Keeping a list of tasks boost accuracy
- For large complex tasks, make two multiple passes, first use CoT and then execute the tasks in later passes

Example: I bought 12 beers, my son drank 8 of them and bought me 6 more, he and his brother then drank 4 each and I drank 2. How many beers do I have left? (actual output)

- To determine how many beers are left, let's carefully follow the sequence of events and track the number of beers consumed or added at each step:
- Step 1: Initial Purchase
  - You bought **12 beers**
- Step 2: Son Drinks 8 Beers
  - The son drank **8 beers**, Remaining beers: **12 - 8 = 4**
- Step 3: Son Buys 6 More Beers
  - The son buys **6 more beers**, Total beers now: **4 + 6 = 10**
- Step 4: Son and Brother Each Drink 4 Beers
  - Son drinks 4, brother drinks 4, Total consumed: **4 + 4 = 8**, Remaining beers: **10 - 8 = 2**
- Step 5: You Drink 2 Beers
  - You drank **2 beers**, Remaining beers: **2 - 2 = 0**
- Final Answer
  - 0 - You have no more beer left, time to buy some more!

Reasoning

Reasoning is where the model has been trained to ask itself questions and reason the answer

- Reading the reasoning can be extremely interesting and often amusing

What is the best short cocktail using Gin?

- Looking up short gin cocktails: Martini, Negroni, Gin and Tonic (though sometimes considered long), Gin Sour, French 75 (sometimes served short), Pegu Club.
- So, the answer could be the Martini, but also mention others. **However, the user asked for the "best,"** so maybe the Martini is the top choice. Alternatively, the Negroni for its balance.
- Alternatively, **the answer could be the "Gin and Tonic" if the user is okay with it being a bit of a long drink**. But the user specified "short," so maybe the Martini is better.
- **I think the best approach is to list a few options**, explain each, and then conclude with the most recommended one.
- In many cocktail guides, the Martini is often cited as a top gin cocktail.
- Alternatively, the "Gin and Grapefruit" is a good option.
- **But to be safe, I'll go with the Martini as the best short cocktail using gin**, but also mention other options. That way, the user gets a comprehensive answer.

All this reasoning is used to formulate a better and response for the user, it's much slower but significantly more accurate

```
<think>
Okay, the user is asking for the best short cocktail using gin. Let me start by recalling what a short cocktail typically means. From what I know, short cocktails are usually served in a rocks glass or a coupe, and they're typically 2-4 ounces, so they're not too strong but still have a good flavor. They're often enjoyed neat or on the rocks, but in this case, the user is asking for a cocktail, so it's probably a mixed drink.

Now, the main ingredient is gin. I need to think of classic gin cocktails that are short. The most popular ones come to mind. The classic gin and tonic is a good one, but it's usually a long drink. Maybe the Tom Collins? Wait, that's also a long drink. Hmm. What about the Negroni? That's a short cocktail, but it's a mix of gin, Campari, and sweet vermouth. But is that considered a short? It's usually served in a rocks glass with ice, so maybe. Then there's the Martini, which is a short cocktail. But a Martini is usually 2-3 ounces, so that fits. But the user might be looking for something that's not too well-known.

Wait, the user said "best short cocktail using Gin." So they might want a recommendation. But what's considered the best? That's subjective, but there are some classics that are widely appreciated. The Martini is a classic, but maybe the user wants something a bit different. The Gin Fizz is another one, but again, that's a long drink.

Wait, maybe the user is looking for something that's a "short" in terms of the glass size, not the strength. So, perhaps the "Gin and Tonic" is a short, but I think it's usually served in a tall glass. Alternatively, a "Gin Sour" could be a short. The Gin Sour is a variation of the classic sour, using gin, lemon juice, and simple syrup. That's a short cocktail.

Another possibility is the "French 75," which is gin, lemon juice, sugar, and champagne. But that's a bit more involved. It's also a bit of a long drink, but maybe served in a coupe.

Wait, the "Gin Rickey" is another one. It's gin, lime juice, and soda water. That's a short, but again, the user might be looking for more complexity.

Alternatively, the "Pegu Club" is a classic gin cocktail with lime and Cointreau. That's a short.

But maybe the user is looking for the most popular or the most recommended. The Martini is definitely a top choice. But some people might consider it too strong. The Negroni is another, but it's more bitter.

I should also consider the balance of ingredients. A good short cocktail should have a nice balance between the gin and the other components. For example, the Martini has gin and vermouth, but the ratio can vary. The Negroni is equal parts gin, Campari, and sweet vermouth.

Wait, the user might not be looking for the most famous, but the best in terms of taste. So maybe the Gin Fizz, but again, that's a long. Alternatively, the "Gin and Grapefruit" which is a variation.

Alternatively, the "Gin Basil Smash" is a short cocktail with gin, basil, lemon, and simple syrup. That's a more modern twist.

But the user might want a classic. So the Martini, Negroni, or the Gin and Tonic. But the Gin and Tonic is more of a long drink.

Wait, maybe the "Gin Sling" is another option. It's a short cocktail with gin, citrus juice, and sugar.

I think the answer should list a few options, but the user asked for the "best." So I need to choose one. But since it's subjective, perhaps present a few and explain why they're good. However, the user might want a single recommendation.

Alternatively, the "Gin and Tonic" is a classic, but it's a long drink. Maybe the user is okay with that. But the question specifies "short," so maybe the answer should focus on that.

Another thought: the "Gin and Soda" is a short, but that's too simple.

Wait, the "Gin Fizz" can be made as a short by using less soda. But traditionally, it's a long.

I think the best answer would be to present a few of the most popular short gin cocktails, explain each, and then perhaps highlight one as the "best." But the user might just want the top recommendation.

In conclusion, the Martini is a classic short cocktail with gin. The Negroni is another. The Gin Fizz, if adjusted to be short. But I need to make sure I'm accurate. Let me check some sources.

Looking up short gin cocktails: Martini, Negroni, Gin and Tonic (though sometimes considered long), Gin Sour, French 75 (sometimes served short), Pegu Club.

So, the answer could be the Martini, but also mention others. However, the user asked for the "best," so maybe the Martini is the top choice. Alternatively, the Negroni for its balance.

Alternatively, the answer could be the "Gin and Tonic" if the user is okay with it being a bit of a long drink. But the user specified "short," so maybe the Martini is better.

I think the best approach is to list a few options, explain each, and then conclude with the most recommended one. But the user might want a single answer. Since the question is phrased as "best," I should pick one and explain why. However, it's possible that the user is looking for a specific one.

In many cocktail guides, the Martini is often cited as a top gin cocktail. But the Negroni is also highly regarded. Another possibility is the "Gin Rickey," which is simple and refreshing.

Alternatively, the "Gin and Grapefruit" is a good option.

But to be safe, I'll go with the Martini as the best short cocktail using gin, but also mention other options. That way, the user gets a comprehensive answer.
</think>
```

The answer?

Classic Martini - A timeless blend of gin and vermouth, the Martini highlights the botanicals of gin. Adjust the vermouth ratio (e.g., "dry" vs. "wet") to suit your preference.

- 2.5 oz gin
- 0.5 oz dry vermouth
- Lemon twist or olive (for garnish)

Negroni - A bold, balanced cocktail with a perfect trio of bitterness (Campari), sweetness (vermouth), and juniper (gin). Served over ice in a rocks glass.

- 1 oz gin
- 1 oz Campari
- 1 oz sweet vermouth

Gin Fizz (Short Version) - A refreshing, fizzy twist on the classic sour. Adjust the sugar to your taste and use a coupe or rocks glass.

- 2 oz gin
- 1 oz fresh lemon juice
- 0.5 oz simple syrup
- Soda water (to top)

Gin Rickey - A light, citrusy cocktail with a crisp finish. Perfect for summer or when you want something zesty.

- 2 oz gin
- 1 oz fresh lime juice
- 0.5 oz simple syrup
- Soda water

French 75 - A sparkling, effervescent drink with a tangy kick. Served in a coupe glass, it’s both elegant and celebratory.

- 2 oz gin
- 0.75 oz fresh lemon juice
- 0.5 oz simple syrup
- Prosecco or champagne (to top)

Multimodal Magic: Vision + Text

Camera Performance

Low read-out noise, high dynamic range.

A lot of human content, on the web, in PDFs etc. have graphics and photos, these are key to the understanding of the content

- Multimodal and vision models can interpret diagrams and photos extremely accurately, they can read graphs, pie-charts and tables
- Using the description of the diagram, graph or photo, the LLM interprets the document similarly to a human taking the image along with the text

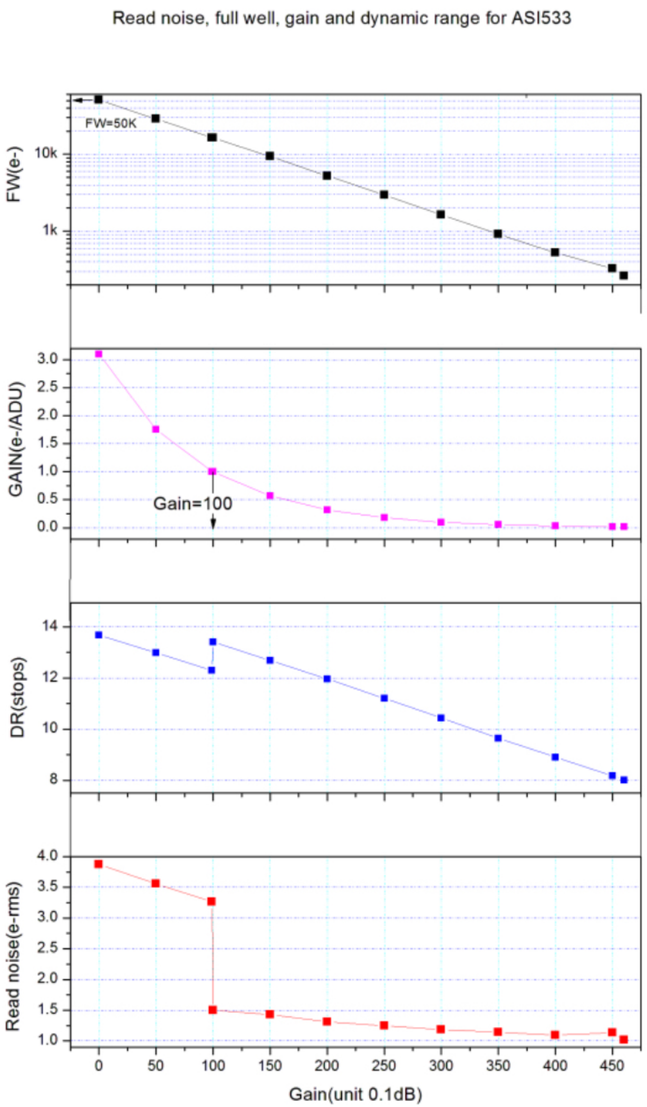
Multimodal models are typically the same speed as text models for pure text but some of the model is used for images so they are not as powerful if used for pure text

- They are also pretty slow with images but, as I'm sure you know images are typically much larger than text so there is more to process

Using the graph, what is the Read noise for gain=50?

- To determine the read noise for a gain of 50, we need to look at the bottom graph in the image, which is labeled "Read noise (e- rms)". Here's the step-by-step process:
  - **Identify the relevant graph**: The bottom graph shows the read noise as a function of gain. The x-axis represents the gain (in units of 0.1 dB), and the y-axis represents the read noise in electron-rms (e- rms).
  - **Locate the gain value**: On the x-axis, find the value corresponding to a gain of 50. Note that the gain is given in units of 0.1 dB, so a gain of 50 corresponds to 500 on the x-axis.
  - **Read the read noise value**: At the point where the x-axis value is 500, look at the corresponding y-axis value. The graph shows a red square marker at this point.
  - **Interpret the value**: The y-axis value at this point is approximately 3.5 e- rms.

Conclusion: The read noise for a gain of 50 is approximately 3.5 e- rms ✅



Demo #2 (Vision): Live Image Q&A

Using the following LLM: **`Qwen2.5-VL-32B-Instruct-bf16`**

With the prompt…

````
"""Detect all the steins of beer in the image and return a JSON array with
the locations.
Example format:
```json
[
  { "bbox_2d": [x1, y1, x2, y2], "label": "Beer"},
  { "bbox_2d": [x1, y1, x2, y2], "label": "Beer"}
]
```

IMPORTANT: Reply with ONLY the JSON array inside a code block.
Do not include any explanatory text before or after the JSON."""
````

Demo

Using th

With the

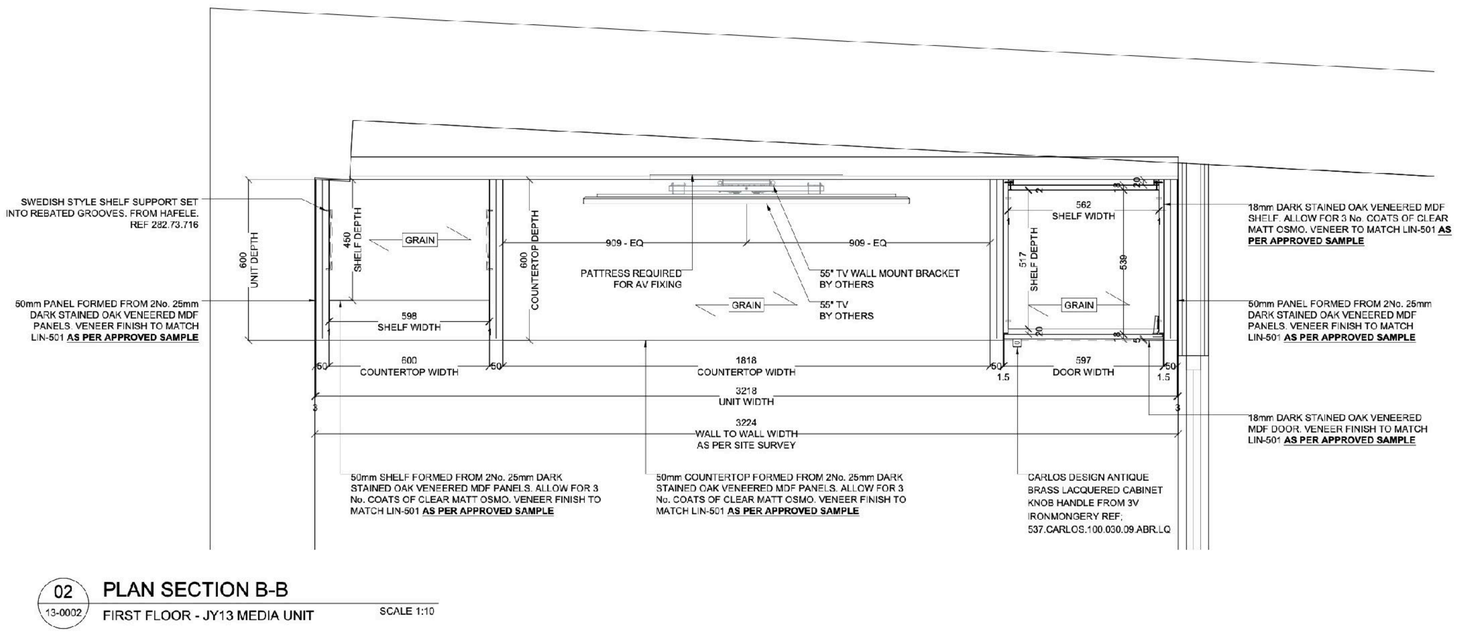
```
Detect a
Example
```json
[
  { "bbo
  { "bbo
]
```
IMPORTAN
Do not i
```

Image to Text - Real user-cases

OCR (Optical Character Recognition) is one of the most important uses, this is usually part of image-to-text

- With OCR we can read entire pages of text (PDFs, images, photos etc.)

We can describe photos, images and even interpret complex diagrams



Retrieval Augmented Generation RAG

This is almost always larger than you imaging, i.e. more useful and on the surface simpler

RAG includes Gen AI with some of the following scenarios, often many at once…

Public Web searches
- Find everything you can about X

Private Web (internal) searches
- Read the company Wiki and find everything about X

Database searches
- Search the database for everything we have on X

Document searches
- Search client contracts, internal documentation, reports, feedback for everything about X

Knowledge Graphs
- Usually combinations of the above but sometimes “walking” through a knowledge graph for more about X

Image searches
- Search data on images, faces, text, descriptions about X

In almost all cases everything can be done faster and better locally - privately too

Processing a large document - RAG

When we have a large PDF or several PDFs, perhaps a book or several books, customer proposals over the years, it is easy to see how we could easily exceed the context of the LLM, several times over in fact

There are a couple of options here, one is to simply summarise the documents before processing them but this eventually meets with the same fate

The other is to cleverly scan through the documents looking for what you need and then feeding that (or those) part(s) in to the LLM

This is called Retrieval Augmented Generation or RAG for short

It doesn’t just relate to documents, recent clients wanted several hundred thousand ZenDesk transcripts, another wanted tweets, hundreds of thousands of them

Documents though are the most common type of original data source, PDFs, Web pages, screen shots, photos, RSS feeds you name it

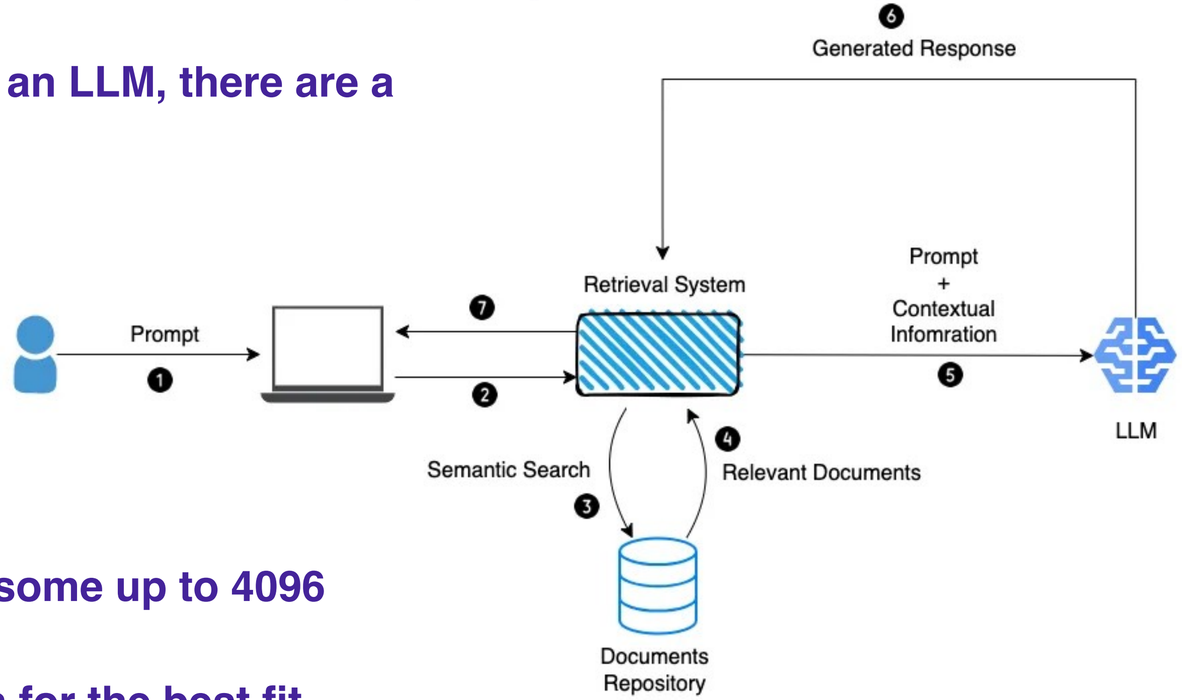
RAG - Retrieval Augmented Generation (from earlier)

Embeddings are a critical part of RAG

- We take a series of documents, “chunk” them into smaller parts and then take an embedding of the chunk
- We usually store the chunk embeddings in a “vector database” which is basically and ordinary database (SQLite, PostgreSQL etc.)
- We then take an embedding of the query and compare each chunk with the query using a cosine similarity

We can, but don’t usually, use the embedding model from an LLM, there are a number of “off the shelf” embedding models

- nomic-embed-text
- mxbai-embed-large
- bge-m3
- bge-large
- all-minilm
- qwen3-embedding (new)
- embeddinggemma (new)



These models are usually 384, 768 and 1024 dimensions, some up to 4096

They vary in capabilities so it’s usually worth testing them for the best fit

- Some are English only
- Some are better at short sentences like X (formally know as Twitter)
- Some are notably faster than others

Chunking

First we read and then chunk the document, chinking is basically, dividing the document into smaller “chunks”

Basic chunk usually gets you 60% there but there are a LOT of techniques to improve chunking

- Brute-force 600 words (or 200 or 1000, just a number)
- Overlapping the chunks, often about 20-25%
- Looking for paragraphs - Usually better than raw chunks
- Identifying sections - usually multiple paragraphs
- Comparing chunks (with embeddings) to see if they chunks have unique information

While the basic chunking usually gets things working, you should not ignore the possibility of improvements with simple optimisation

Chunking is relatively quick and done once, spend more time in this phase and you will get better results

- Sometimes providing the previous and next chunk improves results
- Often more than one embedding can be used for better results
- A reranker also improves results in complex cases

The best results are achieved with the addition of knowledge graphs (linking chunks)

Embedding

We've covered embedding but this is the key to RAG

Choose the wrong embedding library and you will get bad results

- A common mistake is to ignore or forget the maximum context length for the embedder, this is often small, sometime just 512-1024 tokens which is why chunks need to be small

Some models are slow, it's the same old - better is slower paradigm

Matryoshka Embedding Models

- Some embedding models are what's know as Matryoshka Models
- It means that the model still works if you only use part of it
- A model could have 4096 dimensions but the first 512 will work almost as well, even 128 will often do the job

Most embedding models are English only - be careful

- If you have non-English text then make sure your model is multilingual

You can usually batch embedding for better performance and, as we've seen, there are multiple techniques for embedding

Finally the LLM

We pass the original question and 6-20 odd chunks to the LLM with instructions saying basically…

```
You are a document processing assistant, please answer the user's question accurately
using the context provided and only the context provided. If there is not enough
information to answer the user's question then say there is not enough information.
Provide an accurate and informative reply to the use but keep the repsonse brief and
basic only on information provided in the context.

User's question: {prompt}

Context: {context}
```

Like all prompts this can be tuned but a simple prompt like this will go a long way

Getting more advanced, if there isn't enough information in the context then the LLM can be instructed to call a tool to construct a better embedding search

More about tools later

RAG - in practice!

We're going to look at embedding and using RAG on Alice in Wonderland

- Feel free to download your favourite book in any language, books too, this works for multiple books

In a nutshell…

- We chunk the document (book)
- Embed the chunks
- Embed the question and perform a cosine-similarity with each of the embeddings of the book
- We take the resulting chunks that match the best cosines and then pass those to an LLM with the original question
- We ask the LLM to answer the question based on the chunks supplied

That's it!

Later we can store the embeddings in a vector store and make it a lot faster

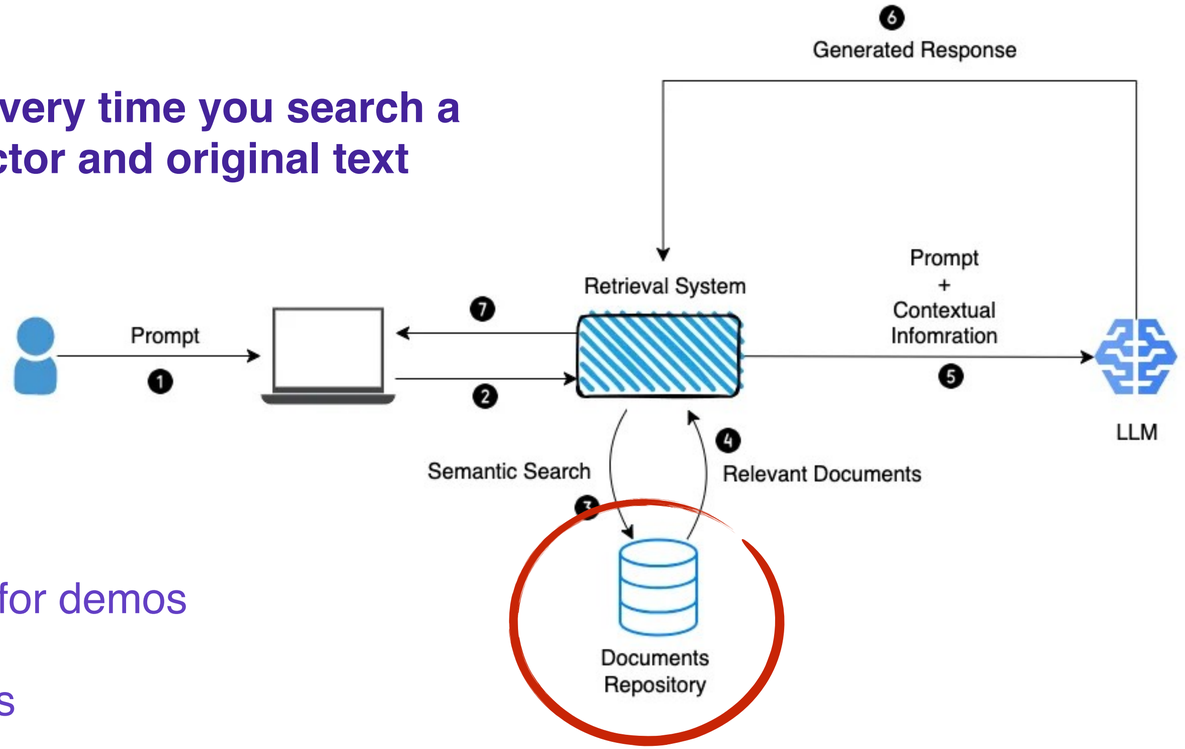
`rag_alice_in_wonderland`

Vector databases - storing embeddings

Using advanced chunking and embedding techniquest could mean that your embedding can take several tens of second

You obviously don't want to do chunking and embedding every time you search a document so we can use a vector database to store the vector and original text

The database will often take your search vector and brute-force search the data using cosine-similarity returning the best matches



Notable Vector databases…

- Pinecone - Closed, SaaS, easy to get started with then $$$
- Chroma - MIT, relatively easy to use and one of the easiest for demos
- Qdrant - Apache 2, good performance
- pgVector - PostgreSQL, basically and extension on Postgres

Storing numbers and text in a database is not rocket science, you can easily knock up a table in SQLite but Chroma and pgVector are easy to use

Embedding

Remember…

- Different languages
- Different context lengths (old ones are usually very short i.e. <500 tokens)
- Embedding can run in Transformers (locally and usually quicker) or remotely via HTTP/S (Ollama / vLLM etc.)
- Embedding can sometime be batched
- Each embedding is different from the others
- Quantisation can reduce quality (just as with LLMs)

If you're not getting good results then

- Check that the data is in the chunks as you expect
- Check that the embedding library you're using is finding the chunks

Do all of this BEFORE you hand the data to the LLM

Explore two more files:

- `rag_alice_in_wonderland_transformers`
- `rag_alice_in_wonderland_chromadb`

Summarisation & Data Extraction

Summarisation is not only for something you can't be bothered to read, it is critical for dealing with the limitations on LLM context size

Summarisation is also useful for RAG, summarising the text is a form of pre-processing and can improve results

Summarisation can also be useful for comments, reporting what you've found, passing on information

Basically, don't underestimate its usefulness

If you take summarisation to the extreme, it is basically data extraction

- Extract the names from the filling text, extract the total price paid, extract the phone number etc.

Data extraction can be more than one thing at a time

- Extract the name, address and phone number from the invoice

Summarisation & Data Extraction in practice

Often what we need from RAG is just the raw facts, we might be contracting something larger and using RAG to fetch the page before extracting the details

We could be creating a CoT for a later pass, first extract the subjects and then iterate through each one

One of the most common uses is to extract data from web scraping, first scrape the page and then extract what we need

With modern LLMs we can combine data extraction and summarisation, usually in that order, first find the data then summarise the results

This is where we often use Knowledge Graphs, we extract not only the data but also the relationships between the data - We then add the data (node) and relationship (edge) to a graph database (e.g. Neo4J)

- The scope of this is beyond the scope of these sessions, it's not really AI, it's the application of AI
- You would probably need a 2-3 day hands-on course to go into building a knowledge graph

A Quick code demo

```
def generate_response(prompt):
    try:
        response = ollama.generate(
            model="qwen3:4b", prompt=prompt, think=False,
            options={"num_ctx": 8192,"temperature": 0.7}
        )
        return response['response']
    except Exception as e:
        print("Error:", e)
        return None

def summarise(text):
    prompt = f"You are a summary assistant. Summarise the following text into very short sentence...\n{text}"
    return generate_response(prompt)

def extract(text, what):
    prompt = (f"You are a concise data extraction assistant. Extract {what} from the following text,"
              f"give the answer only, nothing else...\n{text}")
    return generate_response(prompt)

def main():
    text = """## Model Card for Magistral-Small-2506
    Building upon Mistral Small 3.1 (2503), with added reasoning capabilities, undergoing SFT from Magistral Medium traces and RL on top.
    Learn more about Magistral in our blog post.
    The model was presented in the paper Magistral.

    Sampling parameters
    Please make sure to use:
    * top_p: 0.95
    * temperature: 0.7
    * max_tokens: 40960
    """

    summary = summarise(text)
    print(f"Summary: {summary}")

    data = "Sampling parameters"
    num_paymernts = extract(text, data)
    print(f"{data}: {num_paymernts}")

if __name__ == "__main__":
    main()
```

Try a different LLM from what you're used to

Take the code in scrape.py and modify it to provide a summary of each of the articles in the GDPR spec.

```
if __name__ == "__main__":
    # Single article scraping
    url = "https://gdpr-info.eu/art-17-gdpr/"
    content = scrape_gdpr_article(url)

    if content:
        print("\n" + "=" * 50)
        print("Preview of scraped content:")
        print("=" * 50)
        print(content[:500] + "..." if len(content) > 500 else content)
        print(f"\nTotal characters scraped: {len(content)}")

    # Example: Scrape multiple articles
    # articles_to_scrape = [22, 23, 24]  # Article numbers
    # base_pattern = "https://gdpr-info.eu/art-{}-gdpr/"
    # multiple_content = scrape_multiple_gdpr_articles(base_pattern, articles_to_scrape)
```

Sentiments

Sentiments are very similar to summarisation and data extraction. Instead of extracting a noun or action you're extracting a sentiment

There are many specialist models, most based on BERT models (Bidirectional Encoder Representations from Transformers)

- These fit somewhere between embedding and LLMs
- The latest model in this space is ModernBERT

BERT models are simple enough that they can be fine-tuned to provide better results

However, this is usually only needed to high performance, LLMs do a great job if performance isn't critical and the are a lot more flexible

Sentiment is not always positive / neutral / negative, it can also be happy, sad, threatening, rude, suggestive…

A Simple Sentiment example

```
def analyse_sentiment(text, model="llama3.2"):
    prompt = f"""
    Analyse the sentiment of the following text and respond with exactly one word:
    'positive', 'neutral', or 'negative'.
    Text: {text}
    Sentiment:
    """

    url = "http://localhost:11434/api/generate"
    payload = { "model": model, "prompt": prompt,"stream": False }

    response = requests.post(url, json=payload)
    result = response.json()

    sentiment = result.get("response", "").strip().lower()
```

A Simple Sentiment example (for weather)

```
def analyse_sentiment(text, model="llama3.2"):
    prompt = f"""
    Analyse the weather forecast of the following text and respond with exactly one word:
    ‘wet', 'dry', or 'changeable'.
    Text: {text}
    Sentiment:
    """

    url = "http://localhost:11434/api/generate"
    payload = { "model": model, "prompt": prompt,"stream": False }

    response = requests.post(url, json=payload)
    result = response.json()

    sentiment = result.get("response", "").strip().lower()
```

The whole thing

Small models are fast but not always reliable when it comes to following instructions

It’s usually best to apply some old fashioned code formatting to make it more reliable

Most of this can be done in a fraction of a second on an average machine because it’s only outputting one token (hopefully)

This sort of code can be used to triage messages

```
import requests

def analyse_sentiment(text, model="llama3.2"):
    prompt = f"""
    Analyse the sentiment of the following text and respond with exactly one word:
    positive', 'neutral', or 'negative'.
    Text: {text}
    Sentiment:
    """

    url = "http://localhost:11434/api/generate"
    payload = { "model": model, "prompt": prompt,"stream": False }

    response = requests.post(url, json=payload)
    result = response.json()

    sentiment = result.get("response", "").strip().lower()

    if sentiment not in ["positive", "neutral", "negative"]:
        if "positive" in sentiment:
            sentiment = "positive"
        elif "negative" in sentiment:
            sentiment = "negative"
        else:
            sentiment = "neutral"
    return sentiment

if __name__ == "__main__":
    texts = [
        "I had a wonderful day today!",
        "The weather is cloudy.",
        "This is the worst service I've ever experienced."
    ]

    for text in texts:
        sentiment = analyse_sentiment(text)
        print(f"Text: '{text}'")
        print(f"Sentiment: {sentiment}")
        print("-" * 50)
```

Exercise

Create your own sentiment analysis using data

Kaggle is an excellent web site for reasonably large datasets, it's ideal for testing

The code on the right will download tweets from Trump and display the first few rows, use this or similar as a data source

WARNING: Please put a `time.sleep(1)` if you are using my Groq license as you will quickly hit the rate limit

- If you're testing it's wise to do this anyway

```
{
  "id":1698308935,
  "link":"https:\/\/twitter.com\/realDonaldTrump\/status\/1698308935",
  "content":"Be sure to tune in and watch Donald Trump on Late Night with David Letterman!",
  "date":"2009-05-04 20:54:25",
  "retweets":500,
  "favorites":868,
  "mentions":null,
  "hashtags":null,
  "geo":null
},
```

```python
import os
import pandas as pd
import kagglehub

def load_kaggle_data(data):
    print(f"Downloading {data} dataset...")
    path = kagglehub.dataset_download(f"austinreese/{data}")
    print(f"Dataset downloaded to: {path}")

    csv_file = None
    for root, dirs, files in os.walk(path):
        for file in files:
            if file.endswith('.csv'):
                csv_file = os.path.join(root, file)
                break

    if csv_file is None:
        raise FileNotFoundError("No CSV file found in the downloaded dataset")

    print(f"Reading CSV file: {csv_file}")
    df = pd.read_csv(csv_file)
    print(f"\nDataset shape: {df.shape}")

    return df

if __name__ == "__main__":
    tweets_df = load_kaggle_data("trump-tweets")

    print("\nFirst 6 rows of the dataset:")
    print(tweets_df.head(6))

    print("\nFirst 2 rows as JSON:")
    print(tweets_df.head(2).to_json(orient='records', indent=2))

    print(f"\nTotal number of rows: {len(tweets_df)}")
```

Structured Response

You may have noticed up to now that a lot of the text coming back from the LLM is rather unstructured

We frequently need to ask the LLM to format the data but it's hit and miss as to whether it works or not

- We want smaller models for speed but they are not as well behaved

There is an option in most LLMs to format the output as JSON (not XML)

- This isn't just Ollama, Claude, OpenAI and others support formatted output based on JSON

OpenAI has moved to "Pydantic" and "Zod" for Python and JavaScript, not there is nothing for Java, C# etc.

There are a few subtly different ways to define the output format, some with example, some with schema

Output however may vary. You will get a lot better and more reliable output using JSON than you will by asking for a generic text formatting

In code

I've had mixed but generally good success with JSON format in Ollama and similar models

- {
  "number": 7,
  "English": "seven",
  "French": "sept",
  "German": "sieben",
  "Chinese": "七",
  "Russian": "семь",
  "Arabic": "سبعة"
  }

Write some formatted output, perhaps one of your earlier code examples

```python
import ollama
import json

def generate_formatted_response(prompt):
    try:
        response = ollama.generate(
            model="qwen3",
            prompt=prompt,
            format="json",  # Only accepts "" or "json"
            options={ "num_ctx": 8192,"temperature": 0.3 }
        )
        return response['response']
    except Exception as e:
        print("Error:", e)
        return None

def main():
    prompt = """List the numbers from 1 to 10 and their names in
    English, French, Chinese.
    Provide the output in this exact JSON format:
    {
      "numbers": [
        {
          "number": 1,
          "English": "one",
          "French": "un",
          "Chinese": "一"
        },
        ...and so on for numbers 1-10
      ]
    }"""
    response = generate_formatted_response(prompt)

    try:
        if response:
            parsed = json.loads(response)
            print(json.dumps(parsed, indent=2, ensure_ascii=False))
    except json.JSONDecodeError:
        print("Received non-JSON response:")
        print(response)

if __name__ == "__main__":
    main()
```

Description in the Schema

One of the really “cool” features of using json is that you can describe what you expect to see in the json field and it seems to “magically” populate it…

```
article_schema = {
    "data": {
         "type": "object",
         "properties": {
             "title": {
                 "type": "string",
                 "description": "The exact title of the article"
             },
             "date": {
                 "type": "string",
                 "description": "The publication date of the article in YYYY-MM-DD format"
             },
        },
         "required": ["title", "date"]
     }
}

system_prompt = f"You are a JSON builder expert. You respond to my input according to the
following schema: {json.dumps(article_schema)}"
```

More code writing

Go back to your data extraction from GDPR and format the output as such…

```
“GDRR”: {
    “article”: “Art. 22”
    “summary”: “The GDPR’s “right to be forgotten” lets individuals demand prompt deletion…”
    “full_article”: “1. Where point (a) of Article 6(1) applies, in relation to the offer of information…”
}
```

If things are looking good then create an array for all the articles

Today's Agenda

- **Summarisation & Data extraction**
  Exercise to work on your own web site
- **Sentiments**
  Not just positive / neutral / negative
- **Structured Response**
  { format="json" }
- **Code Generation (Python, C#, SQL etc.)**
  complete, insert, instruct
- **Tool calling**
- **Agentic systems**
- **Testing LLMs**

Code Generation

With everything you've learnt so far and the way you've no doubt been using the models you will have realised that generating code is as simple as telling the LLM you want code to do …

There are three types of code generation…

Complete

- Basically where you provide some existing code and it continues to write, this is typically what you have in code completion

Insert

- This is more complex, the LLM need to know what is before and after the current position, it's called "**Fill-in-the-middle**"
- Some LLMs, code specific models have tags build in to handle this

Instruct

- This is typically how you use Claude or ChatGPT with a "write me some code to…"

Let’s quickly cover the scene for code generation

The best leaderboard for code generation is from Aider:

`https://aider.chat/docs/leaderboards/`

With the release of Claude Sonnet 4.5, this will hopefully see Claude back on the top again

If you’re not wiring SoTA (State of The Art) code then look at using something else, these are all excellent…

- DeepSeek-R1
- Qwen-coder
- DeepSeek-coder
- Yi-Coder
- Codestral

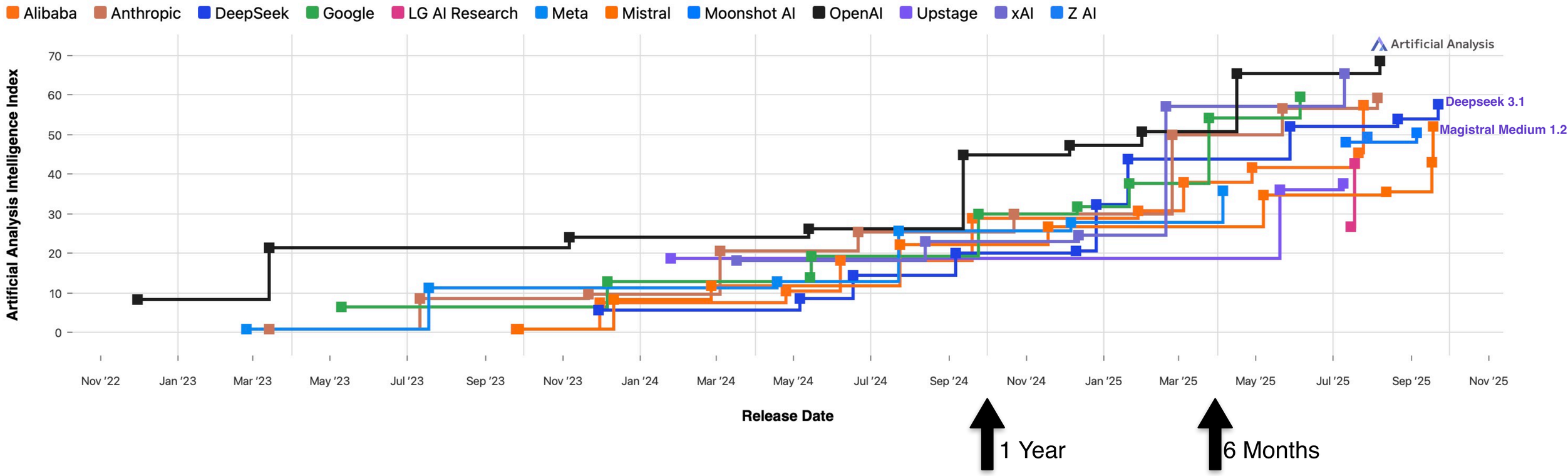
By the end of the month things will have changed and there will be new models

Model	Percent correct	Cost	Command	Correct edit format	Edit Format
gpt-5 (high)	88.0%	$29.08	`aider --model openai/gpt-5`	91.6%	diff
gpt-5 (medium)	86.7%	$17.69	`aider --model openai/gpt-5`	88.4%	diff
o3-pro (high)	84.9%	$146.32	`aider --model o3-pro`	97.8%	diff
gemini-2.5-pro-preview-06-05 (32k think)	83.1%	$49.88	`aider --model gemini/gemini-2.5-pro-preview-06-05 --thinking-tokens 32k`	99.6%	diff-fenced
gpt-5 (low)	81.3%	$10.37	`aider --model openai/gpt-5`	86.7%	diff
o3 (high)	81.3%	$21.23	`aider --model o3 --reasoning-effort high`	94.7%	diff
grok-4 (high)	79.6%	$59.62	`aider --model openrouter/x-ai/grok-4`	97.3%	diff
gemini-2.5-pro-preview-06-05 (default think)	79.1%	$45.6	`aider --model gemini/gemini-2.5-pro-preview-06-05`	100.0%	diff-fenced
o3 (high) + gpt-4.1	78.2%	$17.55	`aider --model o3`	100.0%	architect
o3	76.9%	$13.75	`aider --model o3`	93.8%	diff
Gemini 2.5 Pro Preview 05-06	76.9%	$37.41	`aider --model gemini/gemini-2.5-pro-preview-05-06`	97.3%	diff-fenced

It’s changing week on week

Prioritise what you need - and test. Test, test, test and test again!

- tool-calling
- multimodal
- multi-lingual
- knowledge
- reasoning
- licensing
- context size
- coding ability

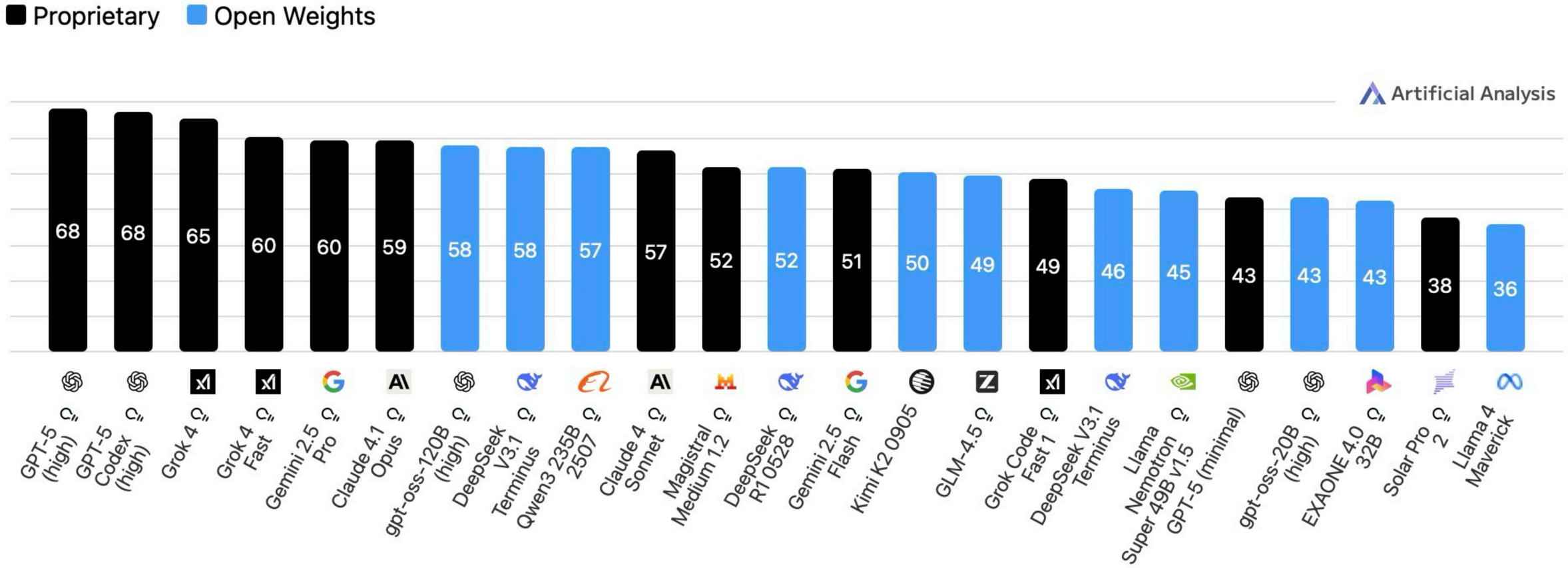


Choosing the Right Model

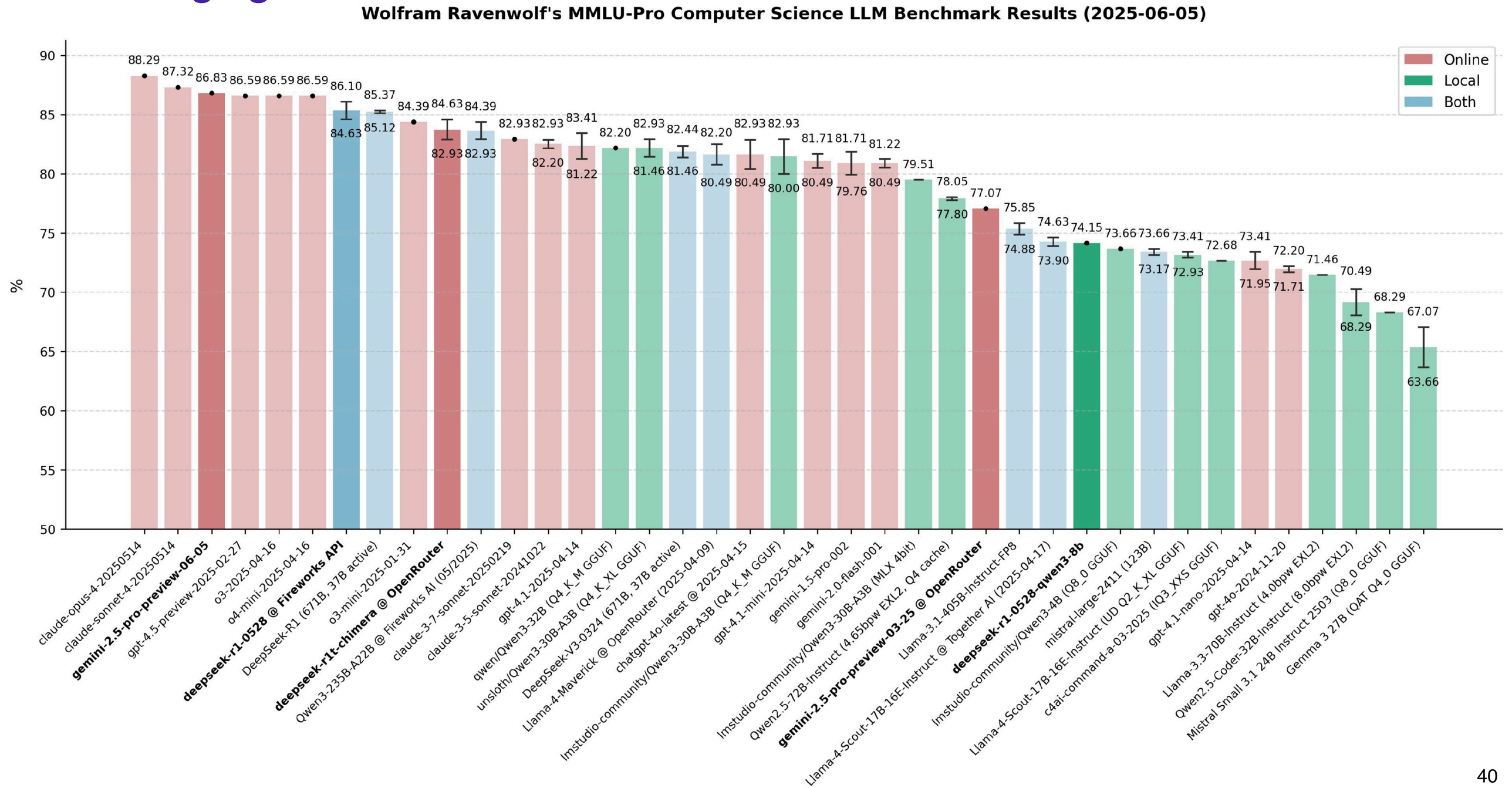
Hopefully you've realised this isn't a simple "which is the best model?"

Basically, newer is almost always better, models improve SIGNIFICANTLY with new releases

- A 3B model today will be better than a 7B model from 6 months ago
- A 7B model today will be better than a 30B model from 6 months ago
- Today's Open Source model will be better than the best proprietary model from 6 months ago



It’s changing week on week



Demo of Claude code

Given that Anthropic released Claude Sonnet 4.5 yesterday, let's take a quick look at Claude Code that now uses Sonnet 3.5

Generating code locally

For the serious code then you can call the models like Claude Sonnet 3.7 to generate code but it's just as easy to use local models or models in Groq and vastly cheaper

```
prompt = """Write a Python function (called calculate_pi) to calculate pi to n
decimal places, where n is a parameter that defaults to 50.
The function should use an efficient algorithm that can generate multiple decimal
places accurately (such as the Chudnovsky algorithm or BBP formula).
Include docstrings, comments, and a simple example of usage.
Only return the code without any explanations outside the code."""

payload = {
    "model": "qwen-2.5-coder-32b",
    "messages": [
        {"role": "system", "content": "You are a helpful assistant. Provide only
the code without explanations."},
        {"role": "user", "content": prompt}
    ],
    "temperature": 0.2,
    "max_tokens": 2000
}
```

Testing

So, you can write code to test something or you can write tests to test you code

- I don’t recommend you write both though

One good way to test the code it run it and call it with expected results

- If the code doesn’t work then feed that back to the LLM and try again - This is agentic

```
decimals = 100
# Test the generated function with 20 decimal places
print(f"\nTesting the generated function with {decimals} decimal places:")
try:
    # First, check if the function name is actually 'calculate_pi'
    if "def calculate_pi" in generated_code:
        function_name = "calculate_pi"

    # Create the execution code with the proper function name
    exec_code = generated_code + f"\n\nprint(f'Pi to {decimals} decimal places
{{{function_name}(decimals)}}')"

    # Execute the code in a controlled environment
    exec(exec_code)
```

Fill in the Middle

Use the code example and get the LLM to generate some interesting code, challenge it with complexity but remember not to expect too many lines, it's not a huge model

If you're feeling up to it, try some fill-in-the-middle

The code on the right is Ollama but it works in the same way with Groq and other models

NB: Fill-in-the-middle or insert is not supported by many modesl

```
from ollama import generate

prompt = """from ollama import generate

def call_llm(prompt):
# User a temperature of 0.7 and context of 4096
"""

suffix = """
    return result

def main():
  reply = call_llm("why is the sky blue?")
  print(reply)
"""

response = generate(
  model='qwen2.5-coder',
  prompt=prompt,
  suffix=suffix,
  options={'num_predict': 512,'temperature': 0,
    'top_p': 0.9, 'stop': ['<EOT>'],
  },
)

print(response['response'])
```

Databases and SQL generation

Generating code from your prompt is one thing but SQL is effectively code and LLMs generate pretty good code

- The also generated pretty good Regex but we'll leave that for another day

If the LLM is made aware of the table structure it will generate SQL of some pretty impressive queries

It is a good idea to include some of the data with the table description as it helps the LLM "understand" the data

payroll.py - Will download a payroll from Kaggle and put it into an SQLite database
payroll2.py - Will open the database and pass the table description with a few records to the LLM with a query

Convert this to Groq or OpenAI and try different queries

Tool Calling - You’re almost there

Now that you’ve seen how to format the response with JSON you can now explore tool calling aka function calling

This is not what is seems

The model is NOT calling tools or functions, it is suggesting tools you can call and helping you with the parameters

This is a huge security advantage, you decide who can call what, when and how

If you ask the model the weather for London, if the model has been told it has a weather tool then it will suggest you call the weather tool with the location “London”. Typically you call the weather with “London”, return the result and then go back to the model with the tool you called (using an ID), the original prompt and the result of the weather

It’s that simple

Tool Calling - The model either supports it or it doesn’t

With many models you can enquire through the API if is supports tall calling

As we’ve learnt earlier JSON is the way the LLM like to exchange information so we use JSON to describe the tools we have available

We can list several tools and the better they are described the more likely the tool is called

You can either describe the function you want to call or you can use the documentation in the function itself

- If of course there is any documentation

You could in theory summarise the function using Gen-AI

```
tools.append({
    "type": "function",
    "function": {
        "name": func_name,
        "description": description,
        "parameters": {
            "type": "object",
            "properties": {},
            "required": []
        }
    }
})
```

Tool Calling - The output

The model tells us we need to call a tool, we can check this on the return of the call to the model

We simply then extract the function name and parameters and then call the function with those parameters

```
"message" : {
  "role" : "assistant",
  "content" : "",
  "tool_calls" : [ {
    "function" : {
      "name" : "understand_tax_codes",
      "arguments" : {
        "tax_code" : "1737L"
      }
    }
  } ]
},
```

```
tool_calls = response['message']['tool_calls']

# Collect tool responses to add to messages
for tool_call in tool_calls:
    # Extract function name
    function_name = tool_call['function']['name']

    # Generate a unique ID if one doesn't exist
    tool_call_id = tool_call.get('id', str(hash(function_name)))

    # Call the function
    if function_name in available_functions:
        function_result = available_functions[function_name]()

        # Add the function call and result to the messages
        messages.append({
            "role": "assistant",
            "content": None,
            "tool_calls": [tool_call]
        })

        messages.append({
            "role": "tool",
            "tool_call_id": tool_call_id,
            "name": function_name,
            "content": function_result
        })

        print(f"Called function: {function_name}")
```

Putting it together

Tool or function calling hits the LLM or an LLM twice, for this reason it's critical that the tool-calling model is quick

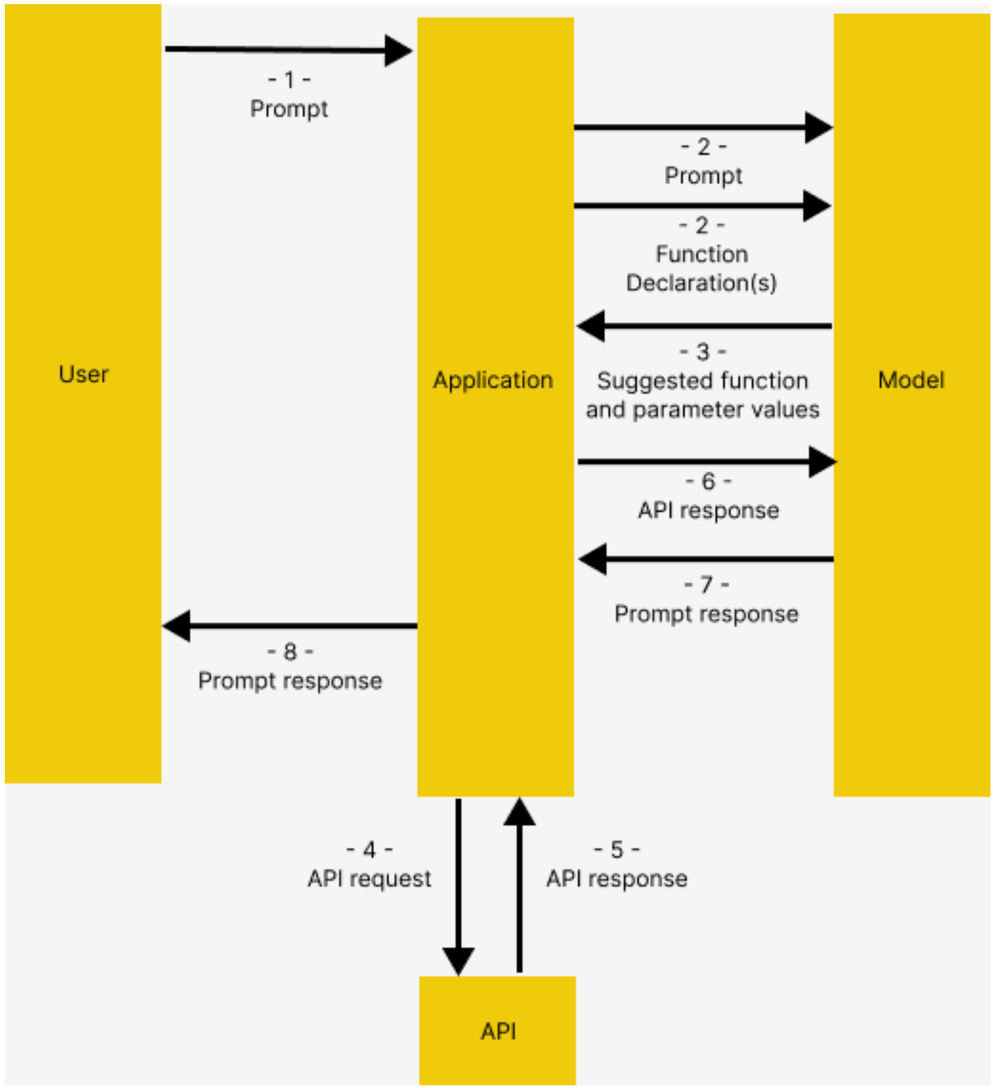
If the parameters need complex processing then things can get slow

Most tool calls are simple

- I need a list of files
- I need to delete a file
- I need to copy a file

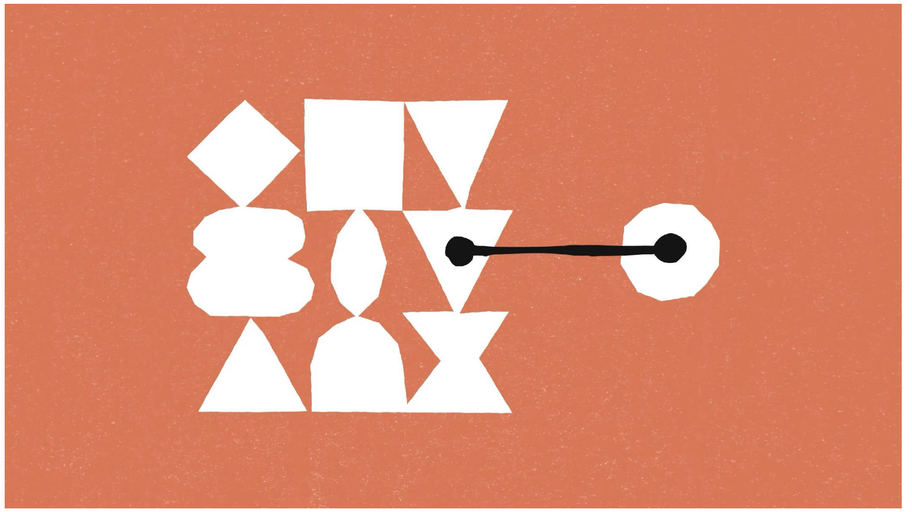
Providing tools to an LLM can make it extremely powerful and seem as if the model has superpowers

- Access a web site
- Get the local news
- Running code - dangerous!



Source: https://blog.christoolivier.com/

Anthropic’s Model Context Protocol (MCP)



MCP is an approach to standardise tool calling

Tools are written generally in Node or Python and publish on a web site so that others can use them

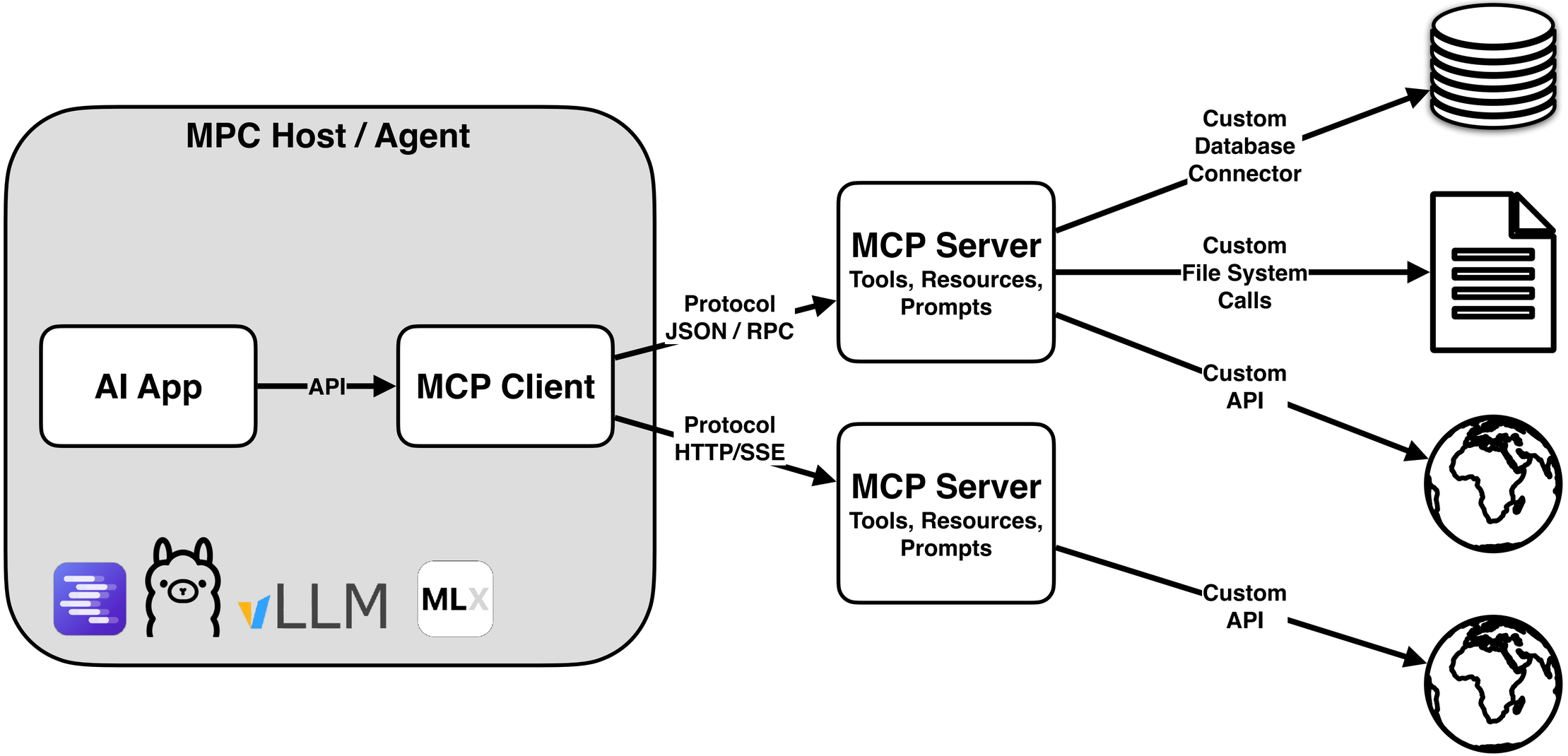
It doesn’t only work in Claude but can be made (easily) to work in Ollan, OpenAI etc., it’s just a matter of making the calls

https://modelcontextprotocol.io/quickstart/user

MCP is an excellent way to build and publish tools for general use

```
{
  "mcpServers": {
    "filesystem": {
      "command": "npx",
      "args": [
        "-y",
        "@modelcontextprotocol/server-
        "/Users/username/Desktop",
        "/Users/username/Downloads"
      ]
    }
  }
}
```

MCP



MCP Example Servers

Data and file systems

- **Filesystem - Secure file operations with configurable access controls**
- **PostgreSQL - Read-only database access with schema inspection capabilities**
- **SQLite - Database interaction and business intelligence features**
- **Google Drive - File access and search capabilities for Google Drive**

Development tools

- **Git - Tools to read, search, and manipulate Git repositories**
- **GitHub - Repository management, file operations, and GitHub API integration**
- **GitLab - GitLab API integration enabling project management**
- **Sentry - Retrieving and analyzing issues from Sentry.io**

Web and browser automation

- **Brave Search - Web and local search using Brave's Search API**
- **Fetch - Web content fetching and conversion optimized for LLM usage**
- **Puppeteer - Browser automation and web scraping capabilities**

Productivity and communication

- **Slack - Channel management and messaging capabilities**
- **Google Maps - Location services, directions, and place details**
- **Memory - Knowledge graph-based persistent memory system**

Official integrations

- These MCP servers are maintained by companies for their platforms:
- Axiom - Query and analyze logs, traces, and event data using natural language
- Browserbase - Automate browser interactions in the cloud
- Cloudflare - Deploy and manage resources on the Cloudflare developer platform
- E2B - Execute code in secure cloud sandboxes
- Neon - Interact with the Neon serverless Postgres platform
- Obsidian Markdown Notes - Read and search through Markdown notes in Obsidian vaults
- Qdrant - Implement semantic memory using the Qdrant vector search engine
- Raygun - Access crash reporting and monitoring data
- Search1API - Unified API for search, crawling, and sitemaps
- Stripe - Interact with the Stripe API
- Tinybird - Interface with the Tinybird serverless ClickHouse platform

Community highlights - A growing ecosystem of community-developed servers extends MCP's capabilities:

- Docker - Manage containers, images, volumes, and networks
- Kubernetes - Manage pods, deployments, and services
- Linear - Project management and issue tracking
- Snowflake - Interact with Snowflake databases
- Spotify - Control Spotify playback and manage playlists
- Todoist - Task management integration

Next Steps

We are at the peak of the AI hype, the "crash" will hit the large propitiatory companies, their suppliers, hosting, funding etc. not th open source models and local hosting

- Small models are the future, own them, they aren't going anywhere

Go local rather than un-reliable, proprietary models

- Faster, Lower latency, Cheaper, Private

Europe is not doing well today but the AI race has just begun

- Start by supporting Mistral

If you want to know the best model, test several, test different versions and quantisations

- As soon as you've finished, everything will have changed - Start again!

Small models and agentic flows are the future, learn to use them an stay in the lead

- Keep an eye on **Embabel** (from Rod Johnson)

Small Language Models are the Future of Agentic AI

Peter Belcak[1] Greg Heinrich[1] Shizhe Diao[1] Yonggan Fu[1] Xin Dong[1]
Saurav Muralidharan[1] Yingyan Celine Lin[1,2] Pavlo Molchanov[1]
[1]NVIDIA Research [2]Georgia Institute of Technology
agents-research@nvidia.com

Abstract

Large language models (LLMs) are often praised for exhibiting near-human performance on a wide range of tasks and valued for their ability to hold a general conversation. The rise of agentic AI systems is, however, ushering in a mass of applications in which language models perform a small number of specialized tasks repetitively and with little variation.

Here we lay out the position that small language models (SLMs) are *sufficiently powerful, inherently more suitable, and necessarily more economical for many invocations in agentic systems, and are therefore the future of agentic AI*. Our argumentation is grounded in the current level of capabilities exhibited by SLMs, the common architectures of agentic systems, and the economy of LM deployment. We further argue that in situations where general-purpose conversational abilities are essential, heterogeneous agentic systems (i.e., agents invoking multiple different models) are the natural choice. We discuss the potential barriers for the adoption of SLMs in agentic systems and outline a general LLM-to-SLM agent conversion algorithm.

It's not AI that is replacing people, it's people using AI who are replacing people who are not


John T Davies
Incept5
https://x.com/jtdavies
https://www.linkedin.com/in/jdavies/