Hands-on GenAI Development Bootcamp Day 2

John Davies

25th November 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()
```

A quick recap of yesterday

We had a quick intro to Python & Ollama

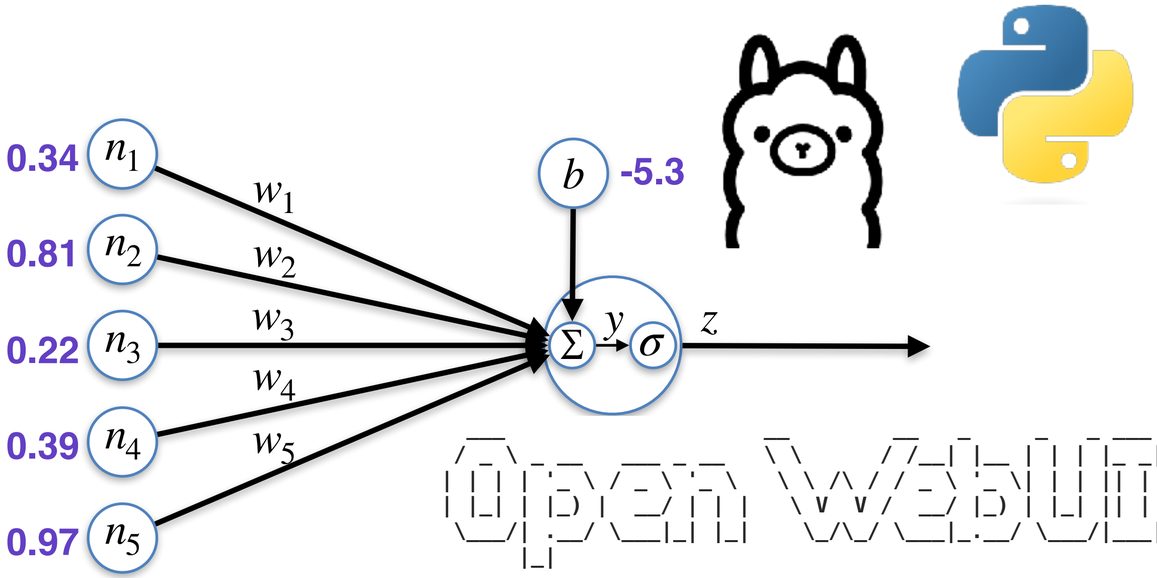
- Keep an eye on Ollama version and new models
- You've also looked at other tools such as Open-WebUI

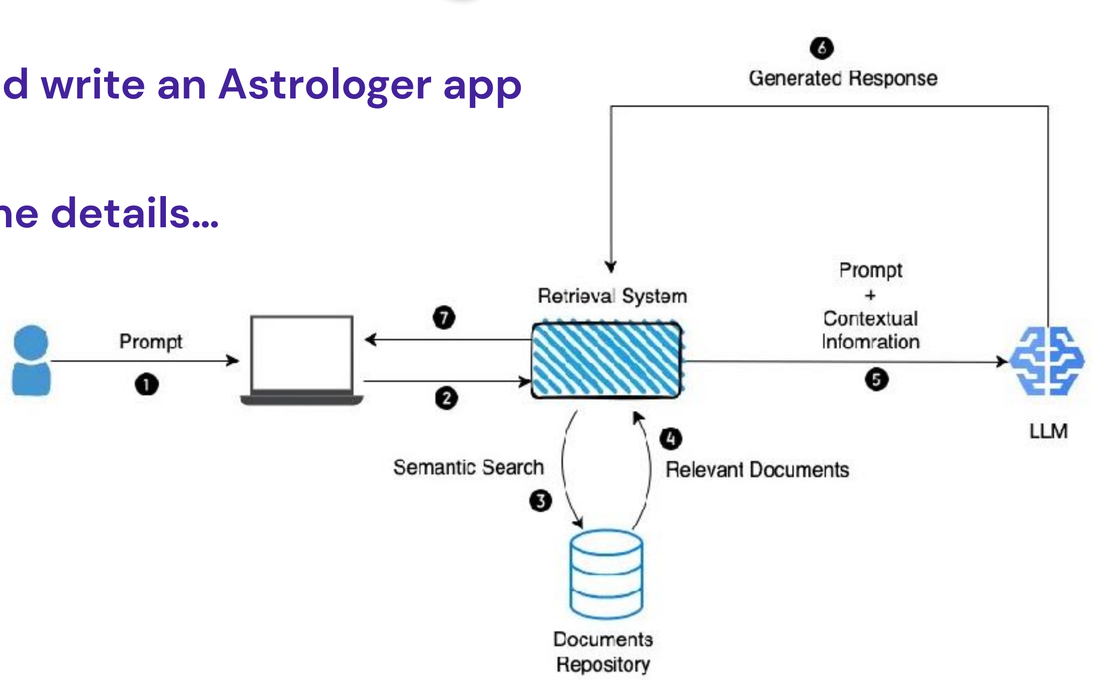
We covered Machine Learning and Neural Networks

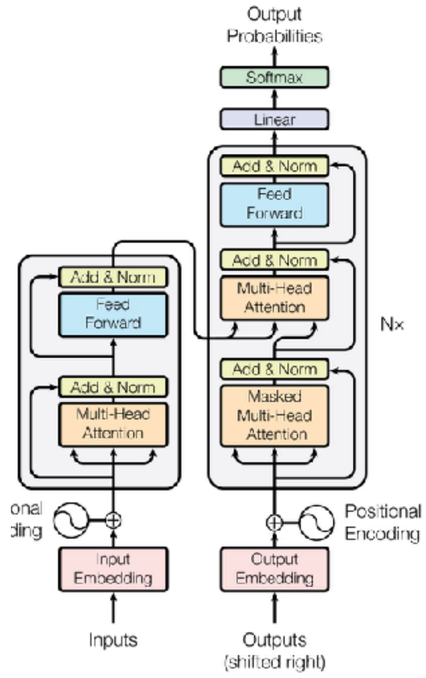
We went into some detail on the LLM and attention heads

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…







From yesterday...

Vision model code (image to text) - coming up next

Demo text to image - coming up next

Text To Speech - coming up

Talk about Vibe coding - Added Claude-code

Text to image – mflux

Using “mflux” (https://github.com/filipstrand/mflux)

```
pip install mflux
```

```
mflux-generate --model dev \
--prompt "a penguin with a speech bubble saying ‘Welcome to MLCon!’" \
--steps 25 -q 8
```


```
mflux-generate \
--prompt "A plate of sausages" \
--model schnell --steps 2 \
--height 1024 --width 1024
```

Deepseek OCR vs Qwen3-vl:4b-instruct

Qwen3-vl:4b-instruct

There is actually a beer here!

Two new bits of code...

- `read_ocr_menu.py` for OCR on a menu
- `find_beer.py` (see photos)

Deepseek OCR

Text to image – ComfyUI

For serious image or video generation I would highly recommend using…

`https://github.com/comfyanonymous/ComfyUI`

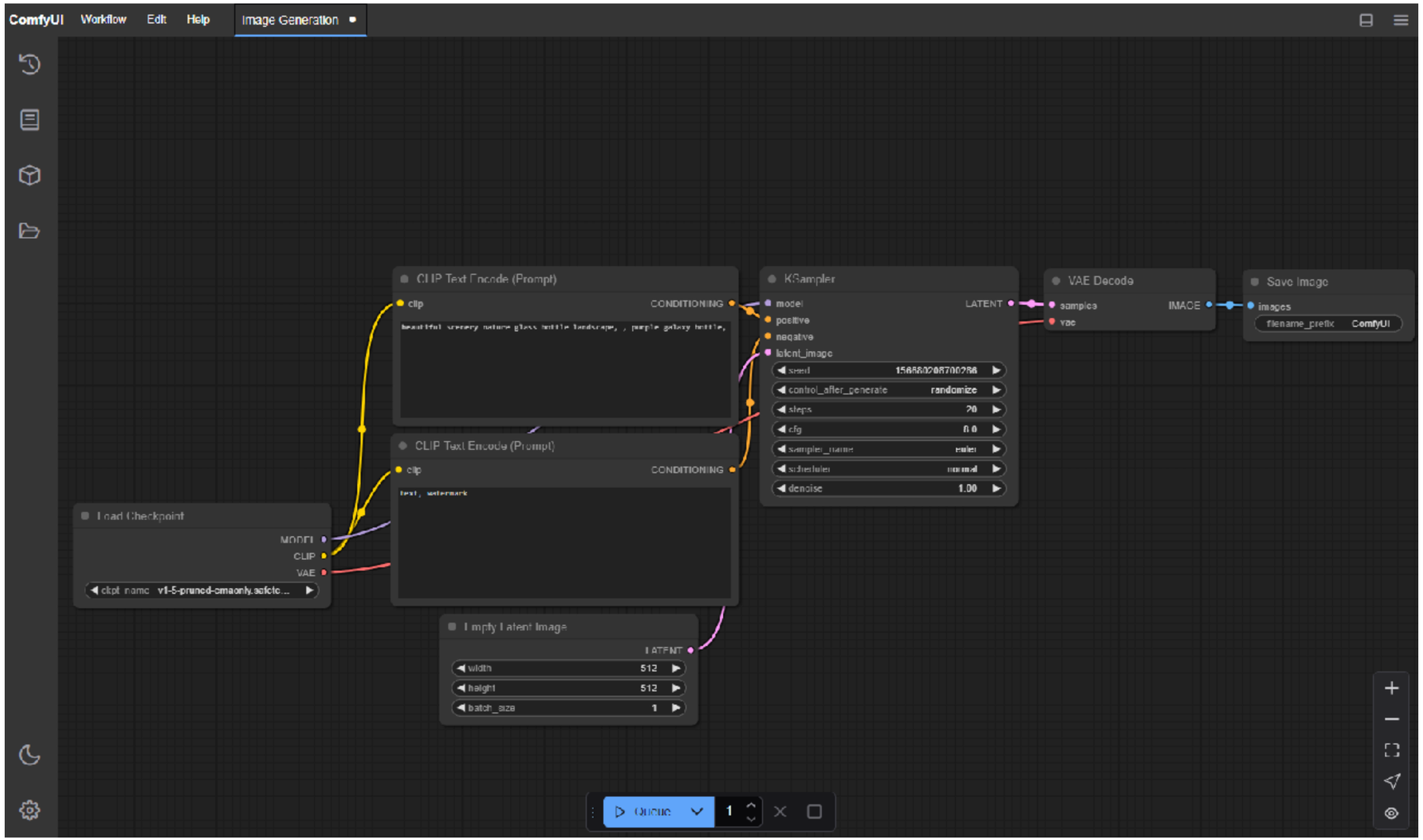


Image to text…

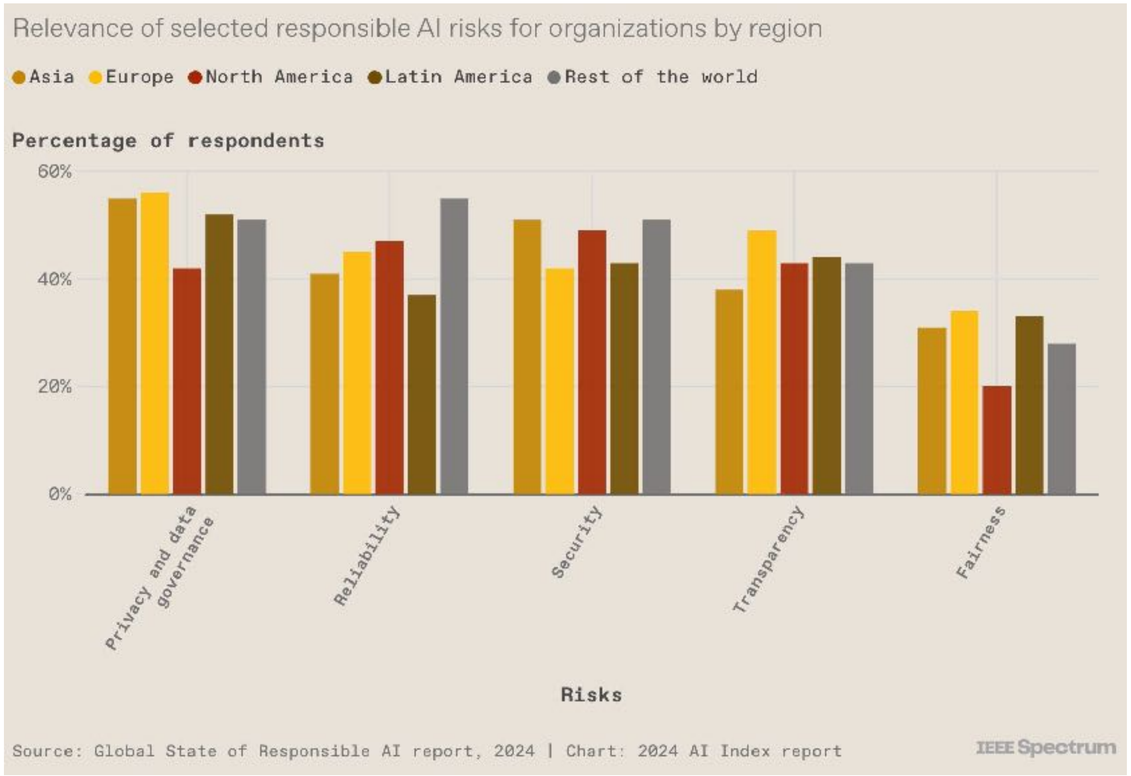
```
ollama run gemma3
>>> Describe this image: /Users/jdavies/dev/mlcon-berlin-2025/day-2/data/IMG_2.jpg
Added image '/Users/jdavies/dev/mlcon-berlin-2025/day-2/data/IMG_2.jpg'
Here's a description of the image you sent:

**Overall:**

The image is a bar chart illustrating the percentage of respondents who identified specific "risks" associated
with responsible AI adoption, broken down by geographic region. The chart focuses on five key risks: Privacy and
data governance, Reliability, Security, Transparency, and Fairness.

**Data Breakdown:**

*   **Regions:** The chart divides the data by:
    *   Asia
    *   Europe
    *   North America
    *   Latin America
    *   Rest of the world
```



In code…

- visual-ollama.py

Text To Speech (TTS)

This can be fun to wrap with a tool-call or MCP, you can then say “Your task is complete, …”

Cartesia is the example used below, Eleven Labs is another popular one but there is no shortage of great models and many are open source - **cartesia_tts_test.py**

```
import os
import sounddevice as sd
import numpy as np
from cartesia import Cartesia

def main():
    # Configuration
    api_key = os.environ.get("CARTESIA_API_KEY")
    if not api_key:
        raise ValueError("CARTESIA_API_KEY environment variable must be set")

    # Text to synthesize
    transcript = "Hi, this is a demo of the TTS, Text to Speech from Cartesia. It's one of the easiest ones to use, Eleven Lab is another example."

    # Voice configuration - using voice ID
    # You can get available voices from: https://docs.cartesia.ai/voices
    voice_id = "6ccbfb76-1fc6-48f7-b71d-91ac6298247b"  # Example voice

    # Initialize client with new API
    client = Cartesia(api_key=api_key)

    # Generate audio using the bytes() method with streaming
    print("Generating audio with Sonic-3...")
```

Retrieval Augmented Generation

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

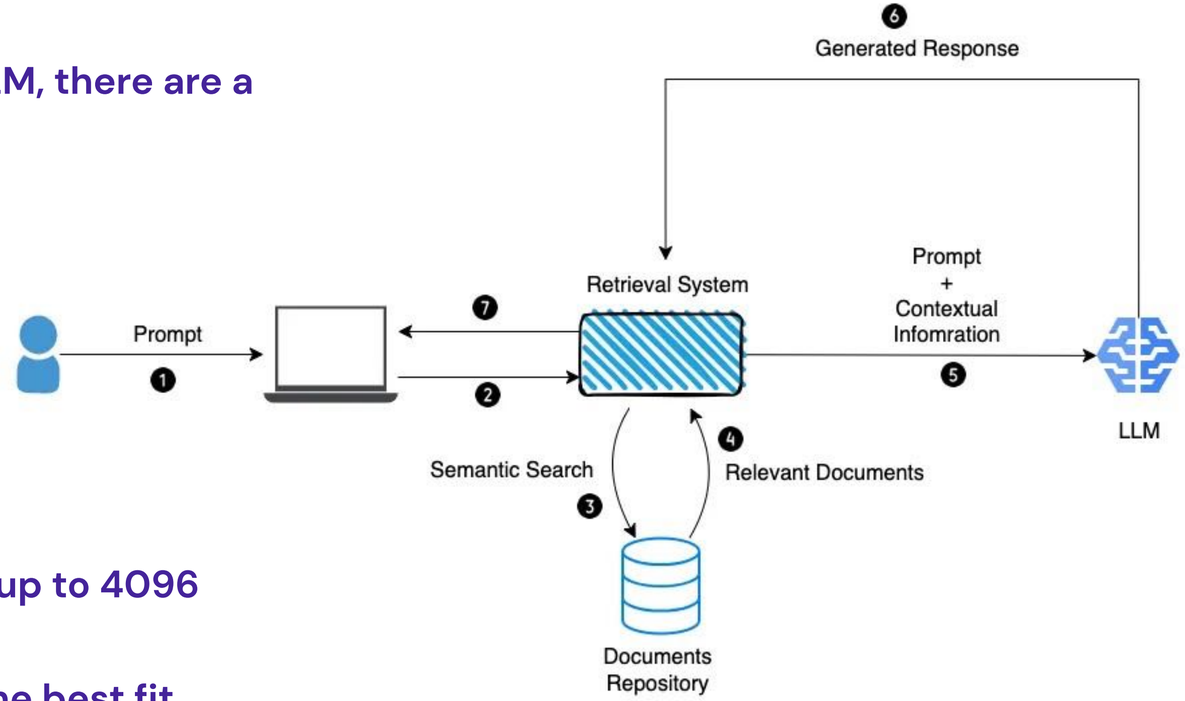
RAG - Retrieval Augmented Generation (from earlier)

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

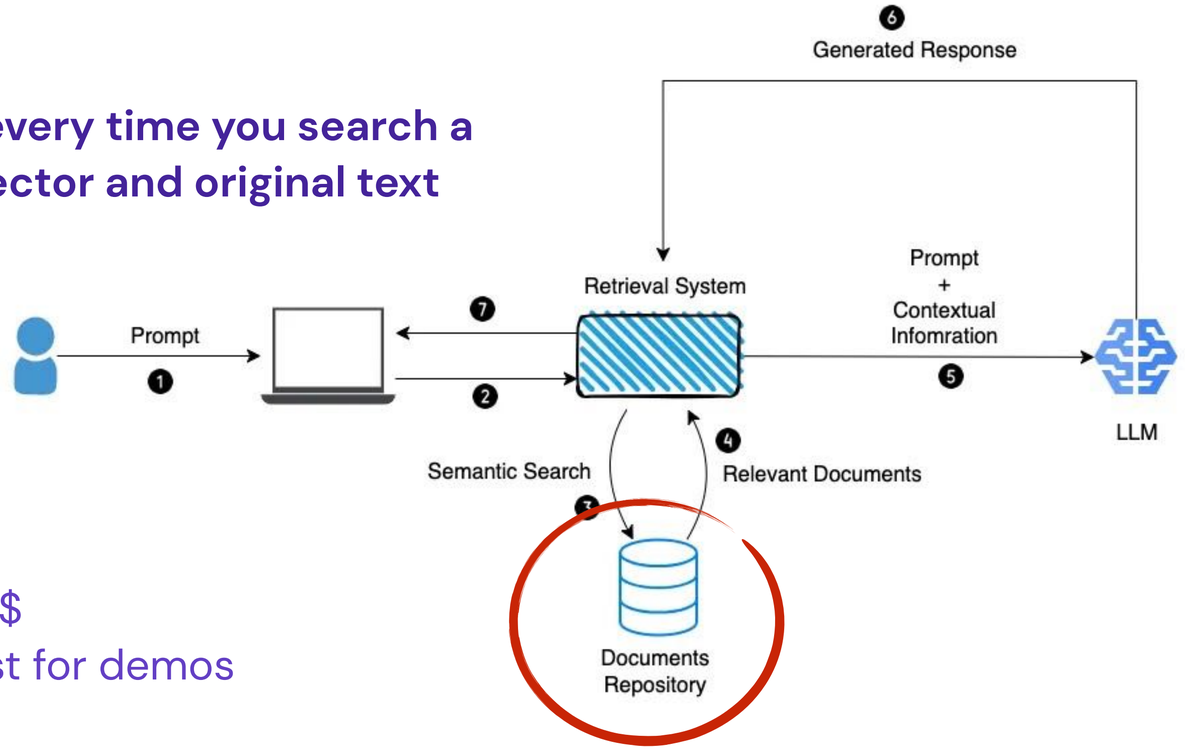
```
rag_alice_in_wonderland
grimm_fairy_tales_rag_demo
```

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`** – **Don't worry about this unless you have the model**
- **`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 (data_extraction_ollama.py)

```python
def generate_response(prompt):
    try:
        response = ollama.generate(
            model="qwen3-vl:4b-instruct", 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="qwen3-vl:4b-instruct"):
    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="qwen3-vl:4b-instruct"):
    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

analyse_sentiment_01.py
analyse_sentiment_02.py

```python
import requests

def analyse_sentiment(text, model="qwen3-vl:4b-instruct"):
    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 – analyse_sentiment_kaggle.py

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"
  }

formatted_response_example.py

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-vl:4b-instruct",
            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

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 Opus 4.5 (a few hours ago), 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

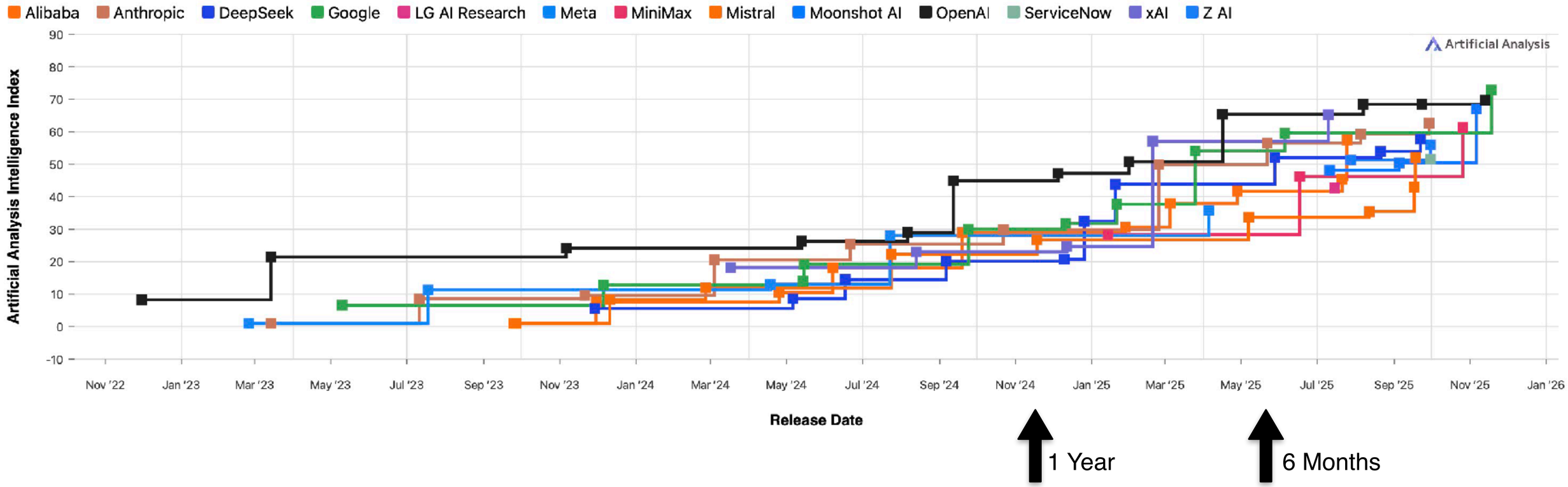
By the end of the month things will have changed and there will be new models

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

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

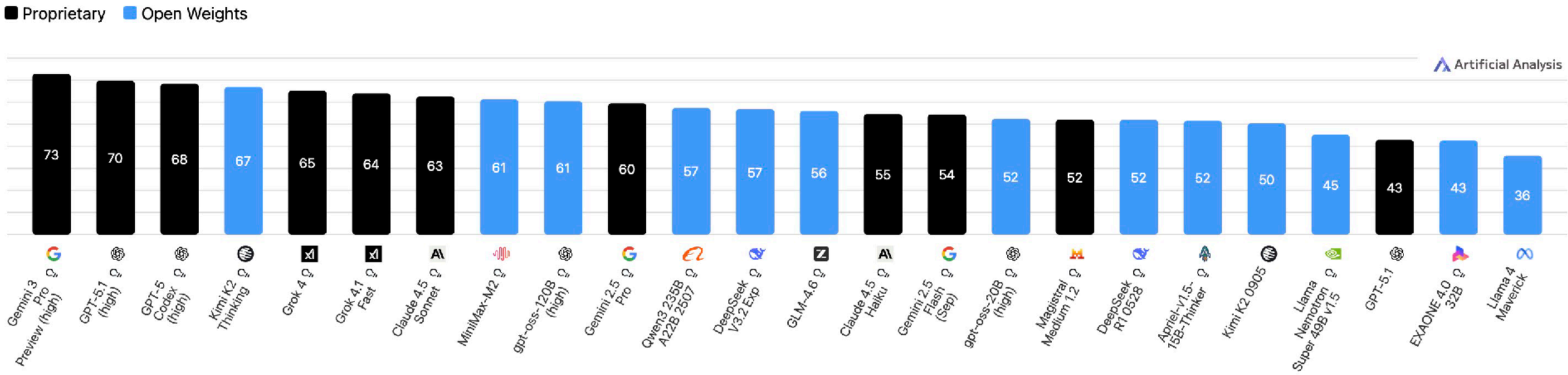


Choosing the Right Model

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



Demo of Claude code

Claude

Today, we're announcing Claude Opus 4.5, the best model in the world for coding, agents, computer use, and enterprise workflows.

Here's what's new on the Claude Developer Platform (API):

- Claude Opus 4.5: The model is a meaningful step forward in what AI systems can do. It's our most efficient model, and is available at $5 input / $25 output per million tokens—making Opus-level capabilities accessible to even more developers and enterprises.

- Advanced tool use (beta): Build agents that can take action with three new capabilities—the tool search tool, programmatic tool calling, and tool use examples. Together, these updates enable Claude to navigate large tool libraries, chain operations efficiently, and accurately execute complex tasks.

- Effort parameter (beta): Control how much effort Claude allocates across thinking, tool calls, and responses to balance performance, latency, and cost.

- Context management capabilities: Enable agents to handle long-running tasks when using tools with the new compaction control SDK helper and reduce token consumption with thinking block preservation, now enabled by default.

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": "codellama:code",
    "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='codellama:code',
  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 plus a few rows to the LLM

Convert this to Groq or OpenAI and try different queries

Purpose of Tool-Calling

Remember when Chat GPT first came out?

- No internet, no web, no up-to-date information

Then they added “The Web”

The web is a tool, ask for something up-to-date or in the news and Chat GPT (or Claude etc.) will call the internet and search

Tool-calling is the ability, as the name suggests, to call tool (a.k.a. functions)

These can be the web, a database, the weather, the current date/time, your recent chat history

However, over the last few months this has gone crazy and almost anything is now a tool-call away with MCP

Tool Schema Structure

Typically a tool call is described in JSON however some models prefer or also handle XML

Either way, the structure is almost the same, you provide the model with the name of the tool (function name), the description, parameters and return type

```
{
   "name": "get_weather",
   "description": "Retrieve weather for a city",
   "parameters": {
      "type": "object",
      "properties": { "city": {"type":"string"} },
      "required": ["city"]
    }
}
```

This tells the LLM that is someone wants the weather for a city it needs to reply (to you) with the name of the function “get_weather” and the name of the city

- It does this by matching the request with the function name and description so the better the description and closer the name to the function the mode likely the call is to work

How Models Handle Schemas

Not all models handle JSON well, they are all getting better as it's becoming critical but you need to…

- Test the models (see code recently used)

Sometimes the full model (fp16) works fine but the quantised version is not so good

- Test the models

Sometimes it works fine for one, two of three tools and then falls down as you hit a dozen

- Test the models

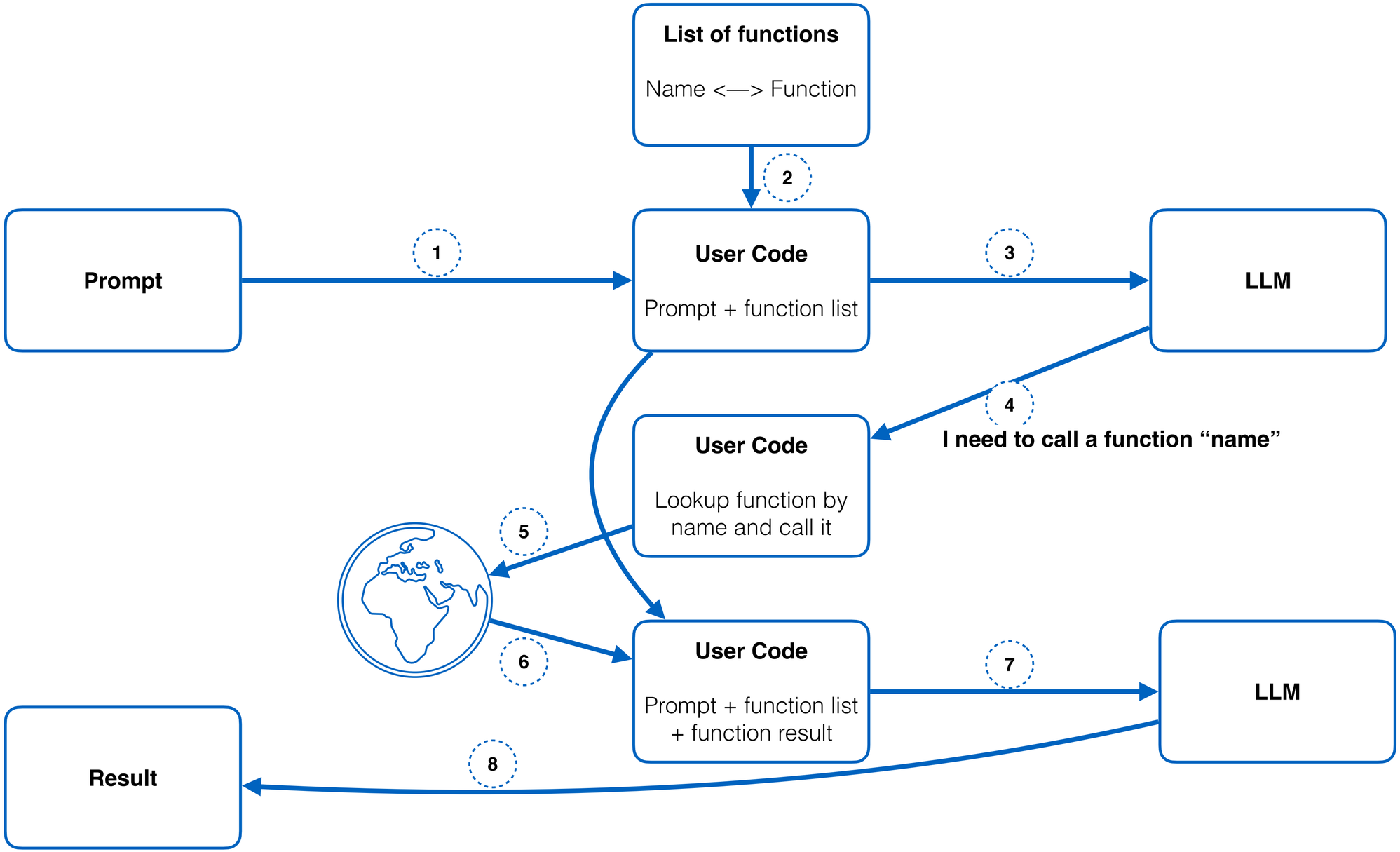
You can use examples in your prompt

- This works extremely well, think of it like teaching other people (or kids), teach by example

Don't make your complex, remember —> KISS

- You're better to take a little longer and do it twice than fail because you made it too complex

Tool-calling flow



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

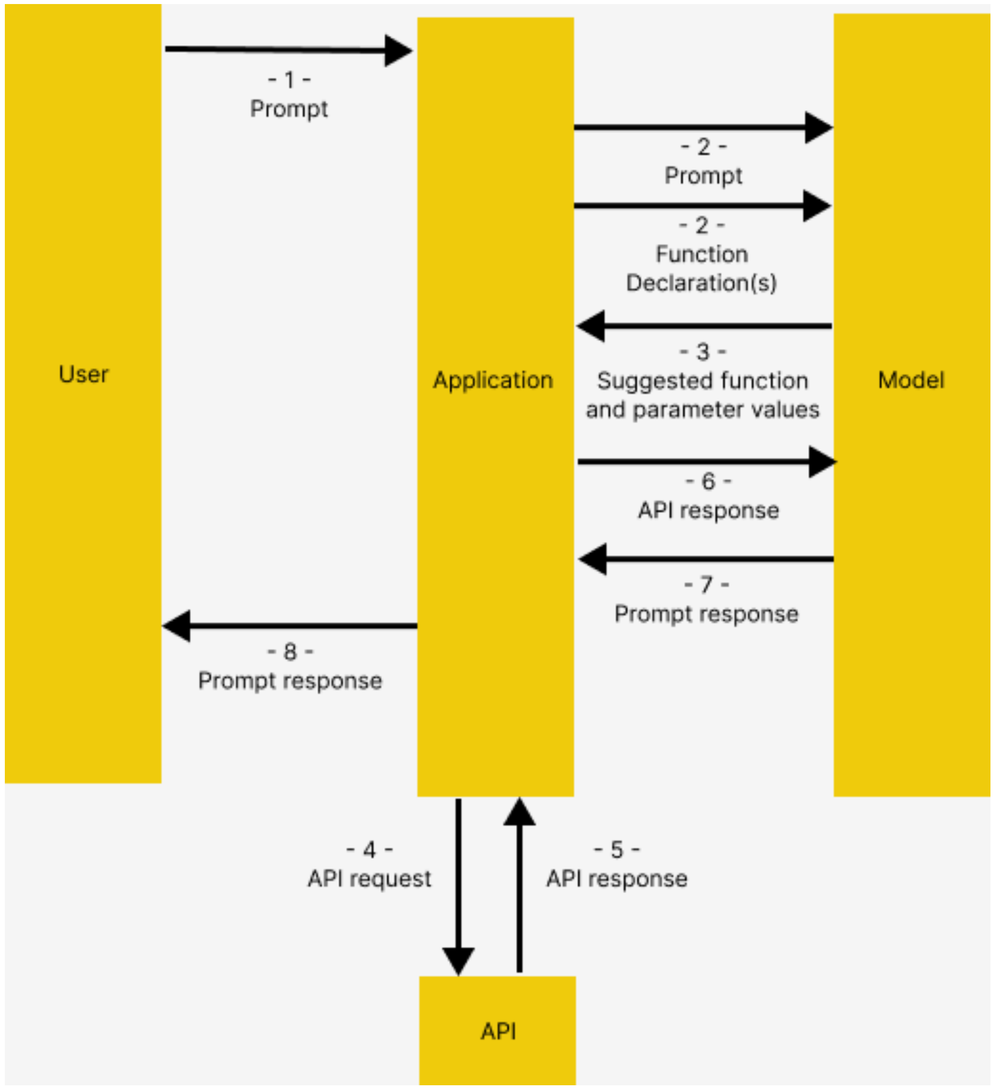
If the parameters need complex processing then things can get slow

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/

Exercise: Tool Design Challenge

Take the function-calling code and add a new function to convert currency

You don’t need to actually call something on the internet but you could return a dummy rate and display the currencies

- For example “What is EUR 50 in USD?” —> results in call to fx() with params…
  - amount = 50.00
  - from = EUR
  - to = USD
- If you check the parameters and return them in the form “50.00-EUR-USD” you can see if it worked

Code Example: Simple Tool Interface

The actual function...

```
private static String convertCurrency(double amount, String from, String to) {
    // Simple exchange rates (hardcoded for demo)
    double rate = switch(from + "-" + to) {
        case "USD-GBP" -> 0.79;
        case "USD-EUR" -> 0.91;
        case "EUR-USD" -> 1.10;
        case "GBP-USD" -> 1.27;
        default -> 1.0;
    };

    double result = amount * rate;
    return String.format("%.2f %s = %.2f %s (rate: %.2f)", amount, from, result, to, rate);
}
```

Defining the tool(s) for the LLM

```
{
  "model": "%s",
  "messages": [
    {"role": "system", "content": "%s"},
    {"role": "user", "content": "%s"}
  ],
  "tools": [
    {
      "type": "function",
      "function": {
        "name": "convert_currency",
        "description": "Convert an amount from one currency to another",
        "parameters": {
          "type": "object",
          "properties": {
            "amount": {"type": "number", "description": "Amount to convert"},
            "from_currency": {"type": "string", "description": "Source currency code"},
            "to_currency": {"type": "string", "description": "Target currency code"}
          },
          "required": ["amount", "from_currency", "to_currency"]
        }
      }
    }
  ],
  "stream": false
}
```

Detecting the tool-call

Unwrapping the result to call the tool

```
if (response1.contains("\"tool_calls\"")) {
    System.out.println("Model wants to call a function!");

    // Check which function is being called
    String functionName = extractString(response1, "name");
    System.out.println("   Function: " + functionName);

    String functionResult;
    if ("convert_currency".equals(functionName)) {
        // Parse function call parameters
        double amount = extractNumber(response1, "amount");
        String from = extractString(response1, "from_currency");
        String to = extractString(response1, "to_currency");

        System.out.println("   Arguments: amount=" + amount + ", from=" + from + ", to=" + to);

        // Execute the function
        functionResult = convertCurrency(amount, from, to);
        System.out.println("   Result: " + functionResult);
```

Anthropic’s Model Context Protocol (MCP)

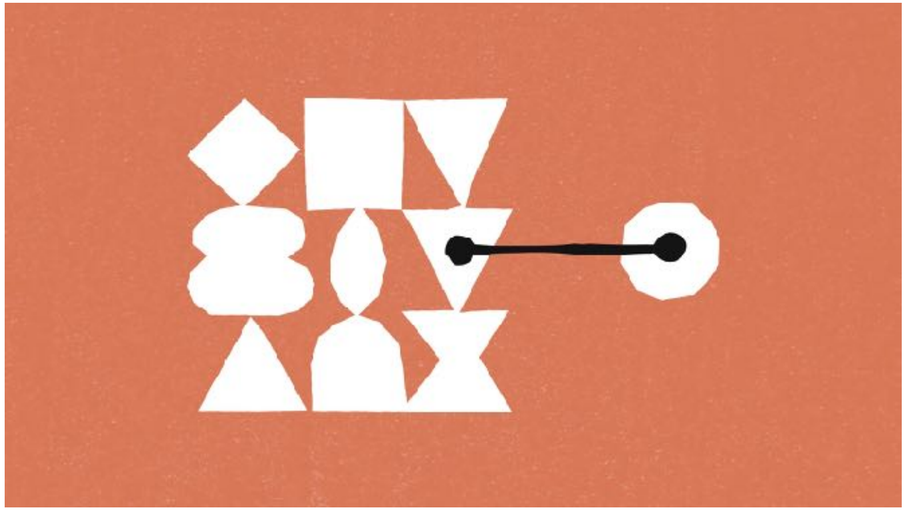
MCP is an approach to standardise tool calling

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

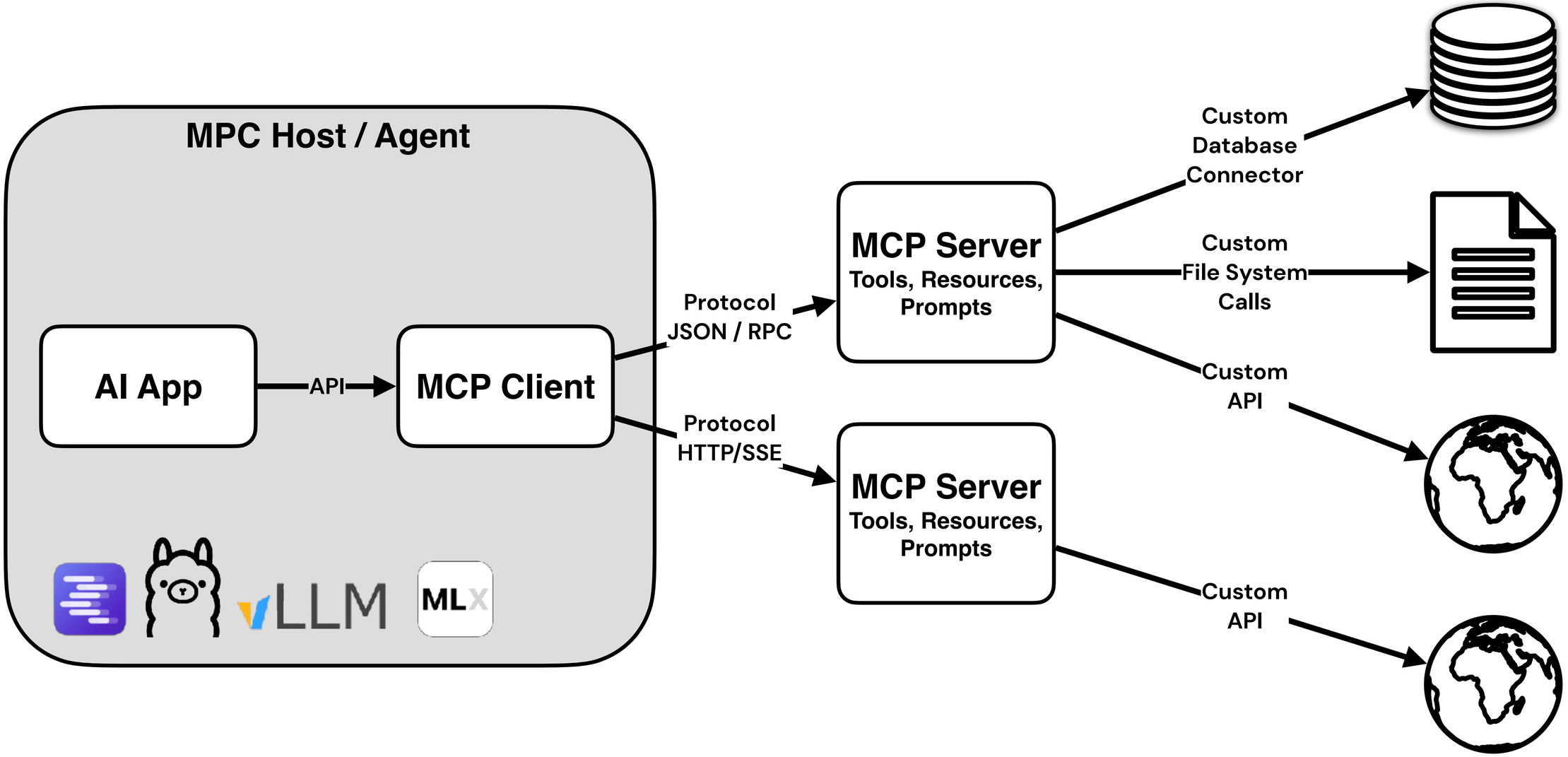
https://modelcontextprotocol.io/quickstart/user

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 galore!

https://github.com/modelcontextprotocol/servers

This reminds me of the early internet, before there were search engines, similar to the early days of Java, there used to be a web site in the 1995 or 1996 where you registered your java code to show others

My point is that there are too many and no quality of service or security

By all means browse but be very careful what you use

MCP Purpose & Scope

Goal: Define a universal interface between LLM agents and external tools or data sources

- Enables consistent discovery, invocation, and description of tool capabilities

Context Exchange: Specifies how session metadata, conversation state, and user context travel with each request

- Prevents “stateless” tool calls - agents can reason over prior context

Interoperability: Common contract across vendors, languages, and frameworks (OpenAI, Anthropic, Embabel, LangChain)

- Promotes plug-and-play tool ecosystems

Developer Benefit: Reduces vendor lock-in-build once, run anywhere that speaks MCP

Enterprise Impact: Unifies compliance, observability, and auditability across toolchains

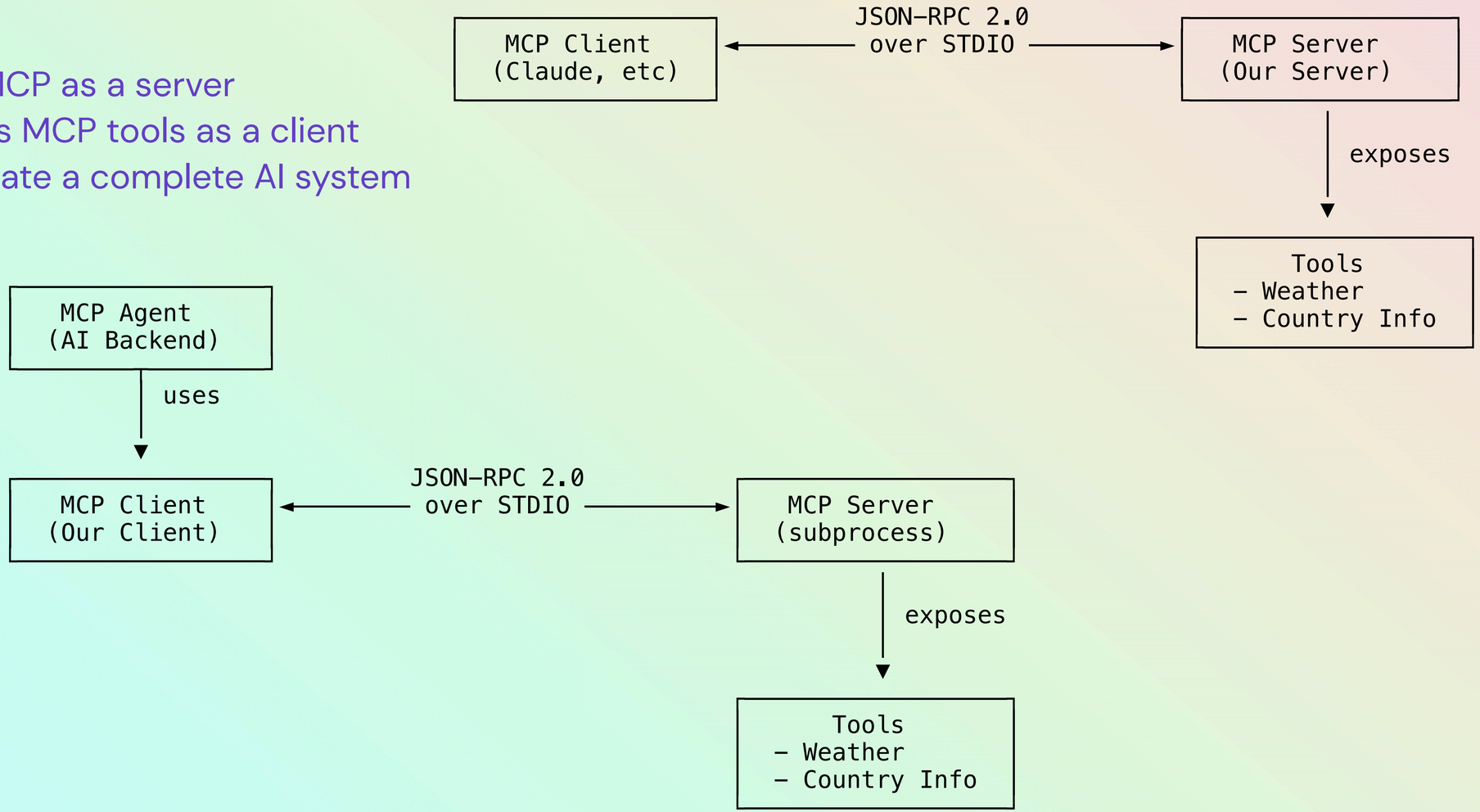
Future Trend: Evolving into a Model-to-Model protocol, allowing chained reasoning across heterogeneous agents

Exercise - MCP Server

This stage introduces the Model Context Protocol (MCP), a standardised protocol for connecting AI applications with external tools and data sources

You'll learn how to:

- Expose tools through MCP as a server
- Build an agent that uses MCP tools as a client
- Connect the two to create a complete AI system

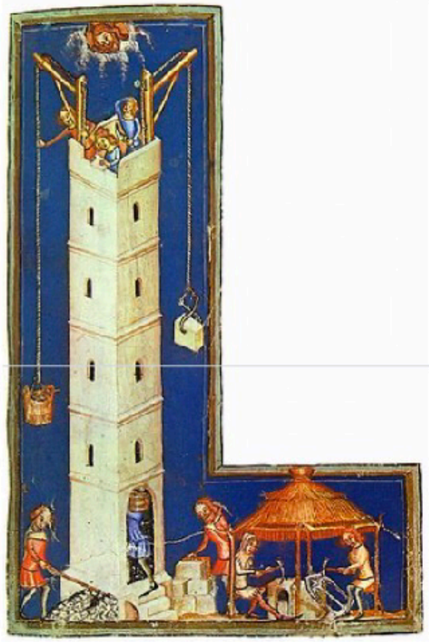


Embabel – Clone the Github repo

https://github.com/embabel

Embabel Agent Framework

KOTLIN JAVA SPRING APACHE TOMCAT APACHE MAVEN CHATGPT JINJA JSON
GITHUB ACTIONS SONARQUBE DOCKER INTELLIJIDEA



Framework for authoring agentic flows on the JVM that seamlessly mix LLM-prompted interactions with code and domain models. Supports intelligent path finding towards goals. Written in Kotlin but offers a natural usage model from Java. From the creator of Spring.

Getting Started

There are two GitHub template repos you can use to create your own project:

- Java template - https://github.com/embabel/java-agent-template
- Kotlin template - https://github.com/embabel/kotlin-agent-template

Or you can use our project creator to create a custom project:

Python vs Java – The Chasm

Pro Python

- Python is great for teaching - it's easy to read, compact (no excessive brackets or semicolons etc.)
- With Python I can read a file, process it and plot is in a graph in just a few line of code
- Python runs everywhere and easy to install - OK packages can be difficult sometimes
- Python libraries can be written in C/C++ for performance - Great for GPU access

Against Python

- It's SLOOOOW 🐢
- It can crash unexpectedly, and debugging stack traces isn't always helpful.
- Hard to manage in production - environments, dependencies, and scaling are tricky.

THE CHASM

Pro Java

- The enterprise workhorse – it replaced mainframes for most mission-critical workloads.
- Virtually every bank and large enterprise runs Java on Linux.
- It's reliable, scalable, and fast – the foundation of production systems.

Challenges with Gen AI in Enterprise

Unavoidable Technical Challenges

- Non-determinism is now the norm, not the exception
- LLM hallucinations create reliability concerns
- Prompt engineering is alchemy, not true engineering
- Cost and environmental implications

Organisational Challenges

- Top-down board mandates without developer buy-in
- Siloed AI teams disconnected from the business
- Greenfield fallacy: ignoring existing systems
- Cannot rollback mistakes like in development
  - **Example: Air Canada chatbot case**
  - **https://www.cbsnews.com/news/aircanada-chatbot-discount-customer/**

MoneyWatch

Air Canada chatbot costs airline discount it wrongly offered customer

By Megan Cerullo

February 19, 2024 / 1:05 PM EST / CBS News

What is Embabel?

JVM-based AI agent framework

- Created by Rod Johnson (founder of Spring Framework) for building production-ready AI agents in Java and Kotlin

Mixes LLM interactions with traditional code

- Actions can invoke LLMs for structured output or execute pure code transformations through strongly-typed domain models

Built on Spring AI ecosystem

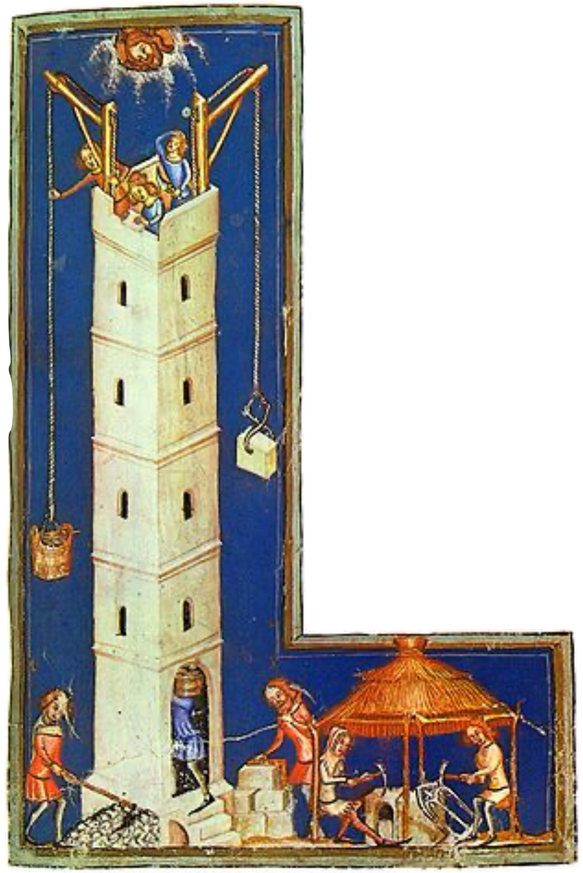
- Leverages Spring's enterprise capabilities while providing higher-level abstractions for complex agentic workflows

Dynamic planning with continuous adaptation

- Automatically replans after each action execution, forming an OODA loop (Observe-Orient-Decide-Act) that adapts to new information

Enterprise-grade and production-ready

- Fully unit testable like Spring beans, with comprehensive testing support and event-driven observability



Embabel - key features

Uses GOAP algorithm instead of LLM-based planning

- Leverages Goal-Oriented Action Planning (from game AI) for deterministic, cost-optimised path finding rather than relying on LLMs for planning

Novel path discovery

- Can combine known actions in unanticipated ways not explicitly programmed, enabling emergent solutions to achieve goals

Strong typing throughout

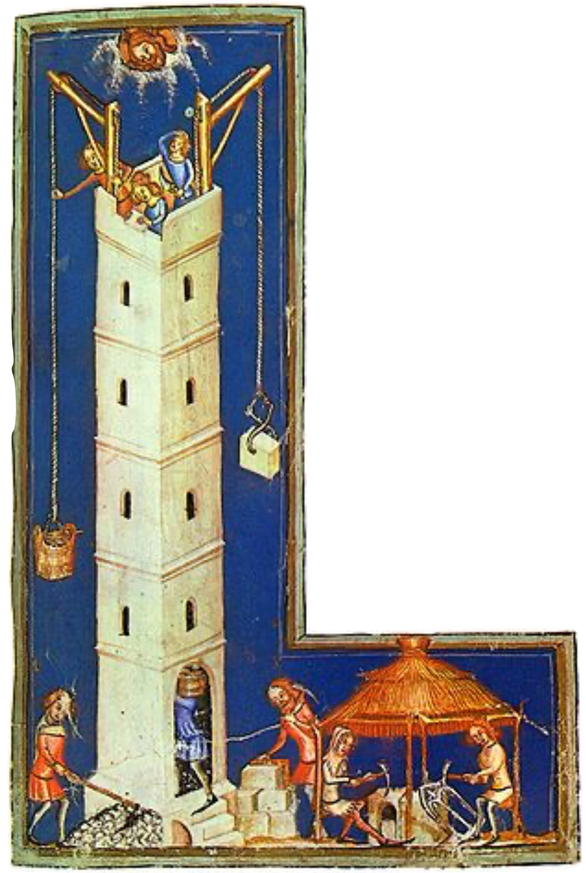
- All LLM interactions are strongly-typed with compile-time checking, refactoring support, and IDE assistance

Reduced unnecessary LLM calls

- Evaluates preconditions and effects to find optimal action sequences, improving efficiency and cost-effectiveness

Extensible without modification

- Can add new domain objects, actions, goals, and conditions without changing existing code, supporting parallelisation when appropriate



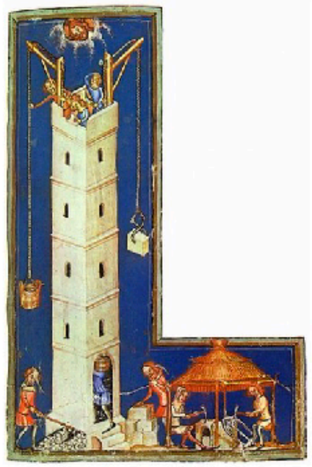
Tripper 😱

The Embabel Travel Planner Agent: `https://github.com/embabel/tripper`

Tripper: Embabel Travel Planner Agent

Build passing

KOTLIN | SPRING | APACHE TOMCAT | APACHE MAVEN | CHATGPT | JSON | HTMX | GITHUB ACTIONS | DOCKER | INTELLIJIDEA



Tripper is a travel planning agent that helps you create personalized travel itineraries, based on your preferences and interests. It uses web search, mapping and integrates with Airbnb. It demonstrates the power of the Embabel agent framework.

Key Features:

- 🤖 Demonstrates Embabel's core concepts of deterministic planning and centering agents around a domain model
- 🌍 Illustrates the use of multiple LLMs (Claude Sonnet, GPT-4.1-mini) in the same application
- 🗺️ Extensive use of MCP tools for mapping, image and web search, wikipedia and Airbnb integration
- 📱 Modern web interface with htmx
- 🐋 Docker containerization for MCP tools
- 🚀 CI/CD with GitHub Actions

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 the 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

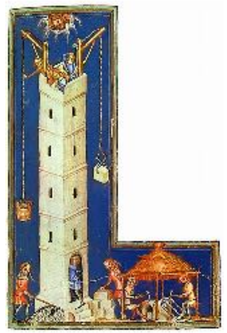
MISTRAL AI_

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 and 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

NVIDIA

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 Davies
john.davies@incept5.com

Vielen Dank

https://x.com/jtdavies

https://www.linkedin.com/in/jdavies/