Lab 5 - LLM

Large Language Models Integration

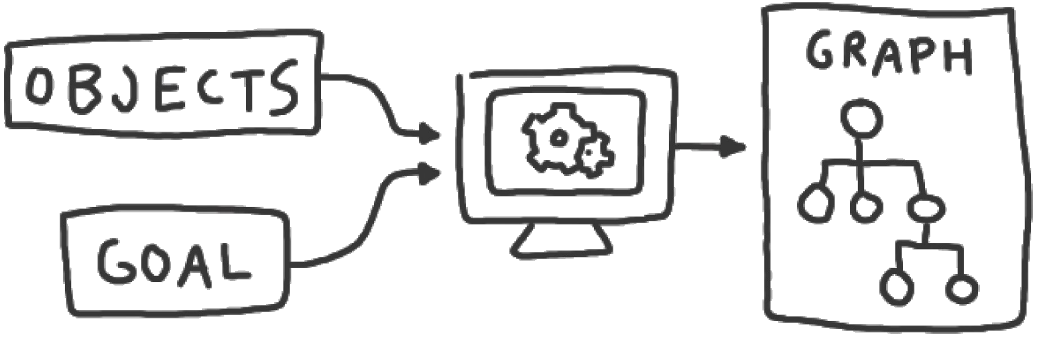
Affordances in a Playground

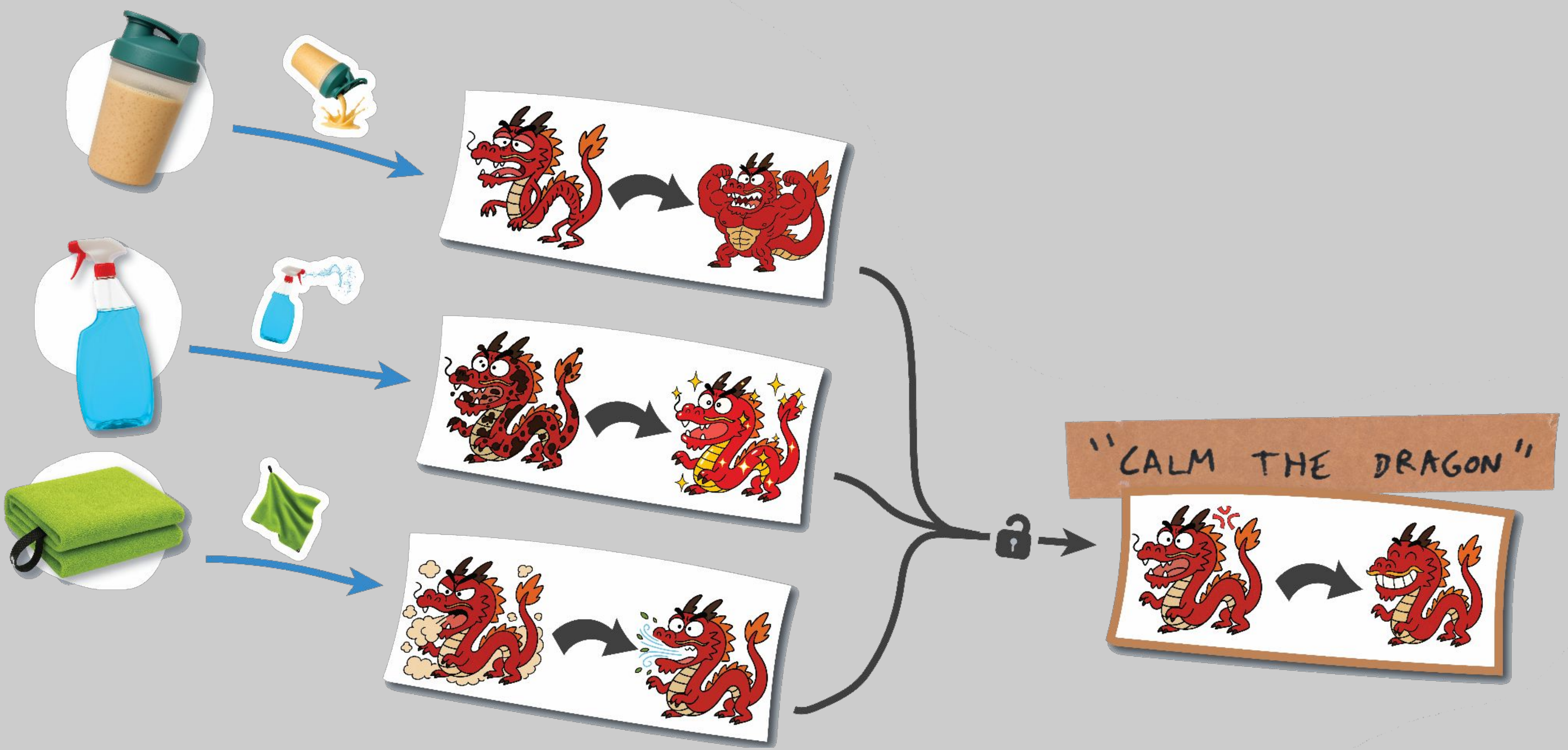
Computer vision has been researching how to do object affordance detection automatically.

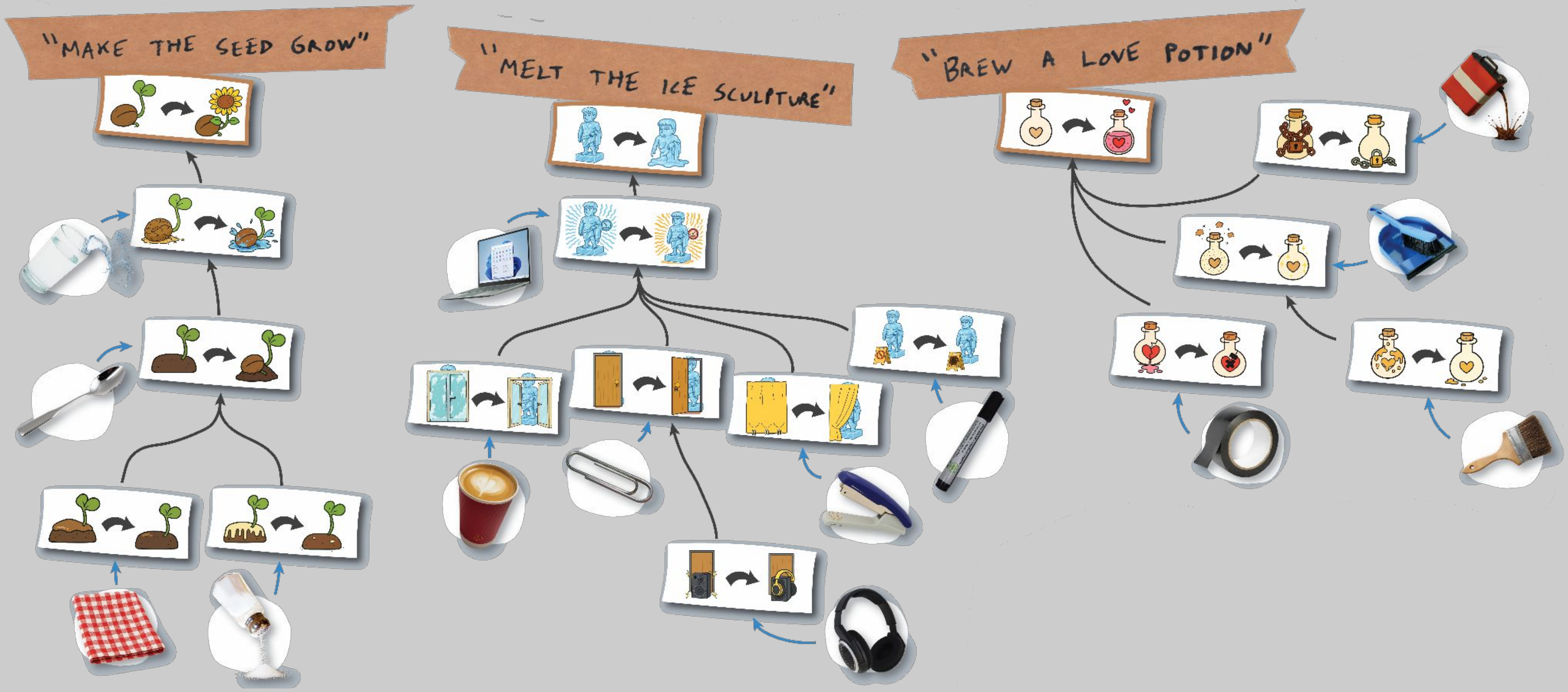
Now, with LLM content generation,

can we generate imaginative interactions from this real world information?


PUZZLE ME THIS:
Generating Playful Challenges by Repurposing Everyday Objects
Alvaro Lopez, Yee Kit Chan, Lung-Pan Cheng

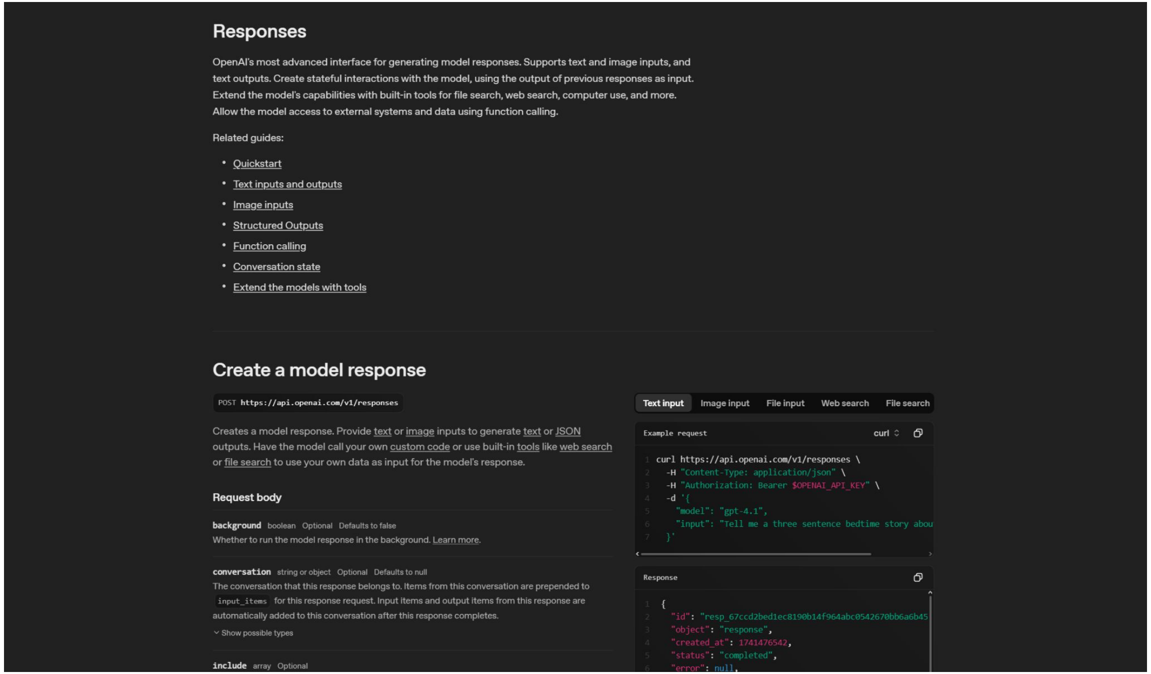

OBJECTS
GOAL
GRAPH


"CALM THE DRAGON"


"MAKE THE SEED GROW"
"MELT THE ICE SCULPTURE"
"BREW A LOVE POTION"

Introduction to GPT API

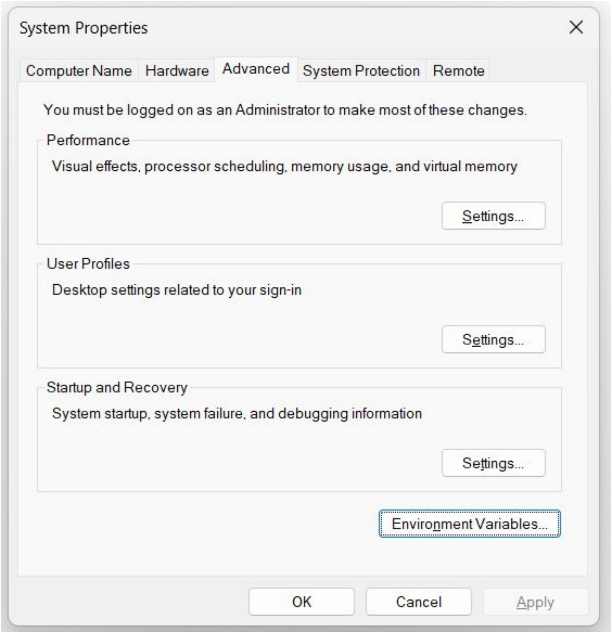
Documentation



[API Reference Documentation](#)

Setting up the API key

- Go to "Edit the system environment variables"
- In the System Properties dialog go to Advanced tab → **Environment Variables...**
- Under **User variables**, click “New”
- Add:
  - Variable name: **OPENAI_API_KEY**
  - Variable value: ***[write your actual API key here]***
- Restart your computer



NEVER write the API key directly to your code:
if you share your code everyone will see your API key

Setting up OpenAI

- On your terminal run:
  - *pip install openai*
- To import the OpenAI library, use:

```
import openai
```

- To access the API key that you set in your environment variables before, use:

```
import os
API_KEY = os.getenv("OPENAI_API_KEY")
```

Initializing the Client

- To create an OpenAI client that you can reuse for every call that you make, use:

```
def initialize_openai():
    openai.api_key = API_KEY
    return openai.OpenAI()
```

- This will set the client's API key to your personal key and return the client

Making an API call

```
def call_openai(prompt, client):
    completion = client.beta.chat.completions.parse(
        model="gpt-4o",
        messages = [
            {"role": "system", "content": "You are a helpful assistant."},
            {"role": "user", "content": "What’s 2 + 2?"},
            {"role": "assistant", "content": "It’s 4."},
            {"role": "user", "content": prompt}
        ],
        temperature=0.5
    )
    return completion.choices[0].message.content
```

- An OpenAI API call looks like this: we send a prompt and obtain a response message from the server

Choosing the model

```
def call_openai(prompt, client):
    completion = client.beta.chat.completions.parse(
        model="gpt-4o",
        messages = [
            {"role": "system", "content": "You are a helpful assistant."},
            {"role": "user", "content": "What's 2 + 2?"},
            {"role": "assistant", "content": "It's 4."},
            {"role": "user", "content": prompt}
        ],
        temperature=0.5
    )
    return completion.choices[0].message.content
```

- **You can choose between different models (*“gpt-5”, “gpt-4”, “gpt-5-mini”, “gpt-4.1”…*) → [More info]()**

Providing the conversation history

```
def call_openai(prompt, client):
    completion = client.beta.chat.completions.parse(
        model="gpt-4o",
        messages = [
            {"role": "system", "content": "You are a helpful assistant."},
            {"role": "user", "content": "What’s 2 + 2?"},
            {"role": "assistant", "content": "It’s 4."},
            {"role": "user", "content": prompt}
        ],
        temperature=0.5
    )
    return completion.choices[0].message.content
```

- At the very least you need the “system” role definition and one “user” prompt
- You can also include previous assistant messages and additional user turns

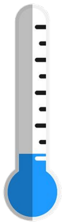
Controlling the temperature

```
def call_openai(prompt, client):
    completion = client.beta.chat.completions.parse(
        model="gpt-4o",
        messages = [
            {"role": "system", "content": "You are a helpful assistant."},
            {"role": "user", "content": "What’s 2 + 2?"},
            {"role": "assistant", "content": "It’s 4."},
            {"role": "user", "content": prompt}
        ],
        temperature=0.5
    )
    return completion.choices[0].message.content
```

- **The temperature for every call can be modified**

Temperature setting

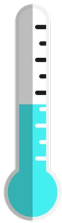
- It controls the randomness or creativity of the LLM



0

as deterministic as it gets:
same input → same output (usually)

use case:
coding, math, structured tasks



0.3 - 0.7

balanced:
a little creativity but mostly reliable

use case:
summaries, explanations, Q&A

0.8 - 1.2

creative:
varied and less predictable

use case:
brainstorming, writing stories, generating ideas

>1.2

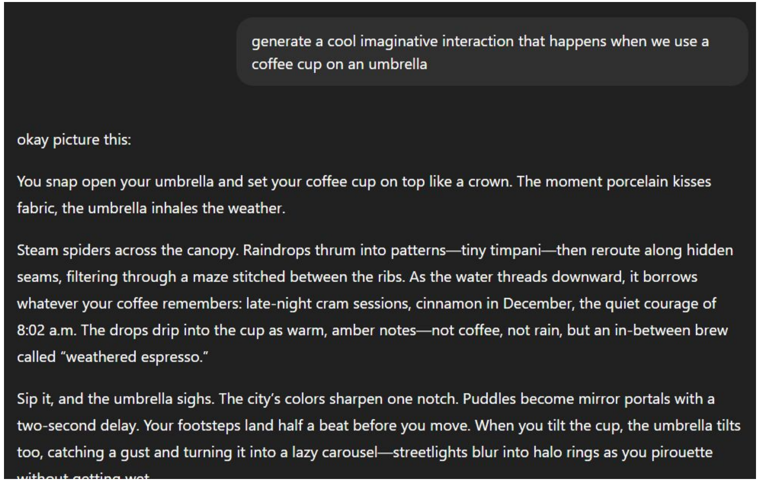
extremely random:
often nonsensical

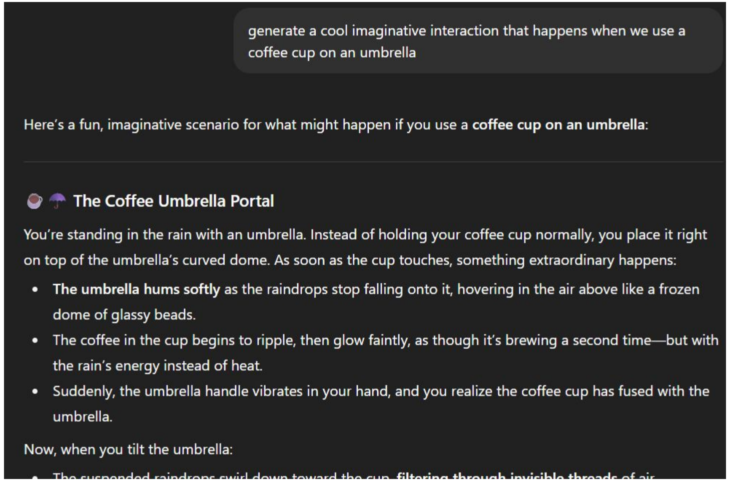
use case:
not very useful, only for very wild idea generation

Building individual prompts

Building our prompt

- If we ask ChatGPT to generate a game interaction, we will get a different result each time and in a random format:

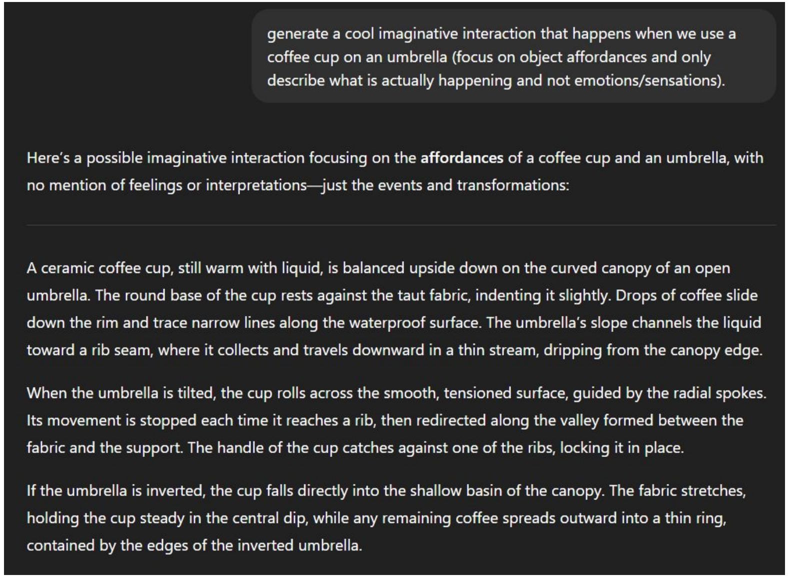




This is hard to use in an actual system: it's hard to predict and hard to control

Prompt engineering

- One thing we can do to improve this is **prompt engineering**
- For example, we can specify the kinds of interactions we want:



generate a cool imaginative interaction that happens when we use a coffee cup on an umbrella (focus on object affordances and only describe what is actually happening and not emotions/sensations).

Here's a possible imaginative interaction focusing on the **affordances** of a coffee cup and an umbrella, with no mention of feelings or interpretations—just the events and transformations:

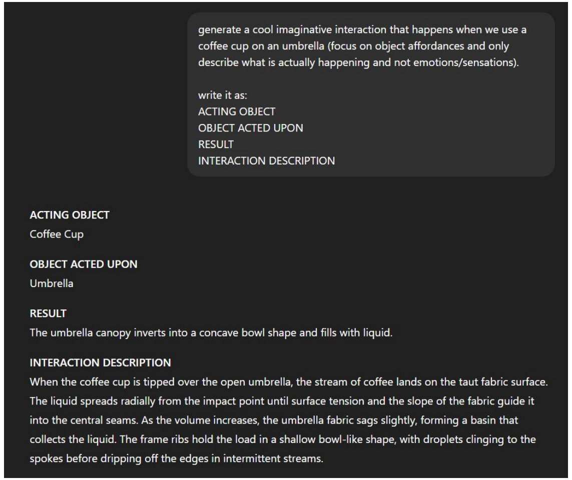
A ceramic coffee cup, still warm with liquid, is balanced upside down on the curved canopy of an open umbrella. The round base of the cup rests against the taut fabric, indenting it slightly. Drops of coffee slide down the rim and trace narrow lines along the waterproof surface. The umbrella's slope channels the liquid toward a rib seam, where it collects and travels downward in a thin stream, dripping from the canopy edge.

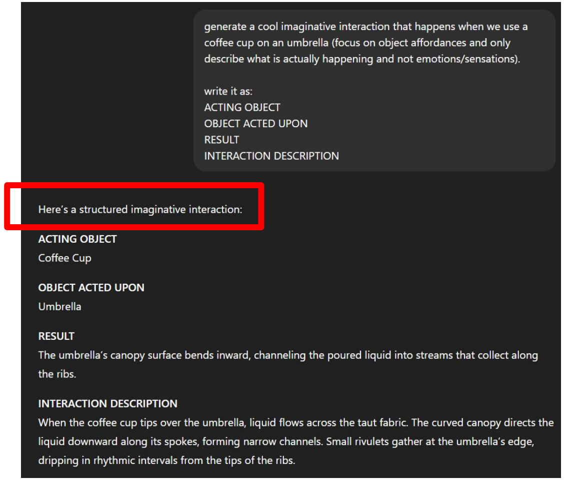
When the umbrella is tilted, the cup rolls across the smooth, tensioned surface, guided by the radial spokes. Its movement is stopped each time it reaches a rib, then redirected along the valley formed between the fabric and the support. The handle of the cup catches against one of the ribs, locking it in place.

If the umbrella is inverted, the cup falls directly into the shallow basin of the canopy. The fabric stretches, holding the cup steady in the central dip, while any remaining coffee spreads outward into a thin ring, contained by the edges of the inverted umbrella.

Prompt engineering

- We can specify a format directly in the prompt, but it will interpret it in a different way every time.
    - Meaning that if we write code that parses this, it will likely **break**
    - **Even if we run the same prompt many times, we don't know if next time it will interpret it in a new way**





Prompt engineering

- Just like in machine learning, we need to avoid **overfitting**
- Specifying concrete examples of what we want or we don't want usually biases the generated content.
    - Instead, we want to find **overall rules** or **constraints** that maintain the generated content in our wanted range, without going into details that narrow its creativity.

generate a cool imaginative interaction that happens when we use a coffee cup on an umbrella (in a small paragraph and only describe what is actually happening and not emotions/sensations).

example: if we use a soap bottle on a portable fan, the fan blows bubbles that can trap enemies

When a coffee cup is poured onto an umbrella, the fabric absorbs the liquid and instantly transforms it into a swirling canopy of steam that spirals outward, forming a protective dome of mist that obscures vision and dampens projectiles passing through.

Now all the interactions will have something to do with water/steam/rainclouds and hindering enemies, instead of using object affordances creatively

Structured outputs

- We can use **structured outputs** to define the structure and types of the generated content.
- This is useful for generating content that can be used in real systems: even if the actual content changes every time, we can predict with 100% accuracy the overall format of that content and write code that can access it without breaking.
- More info

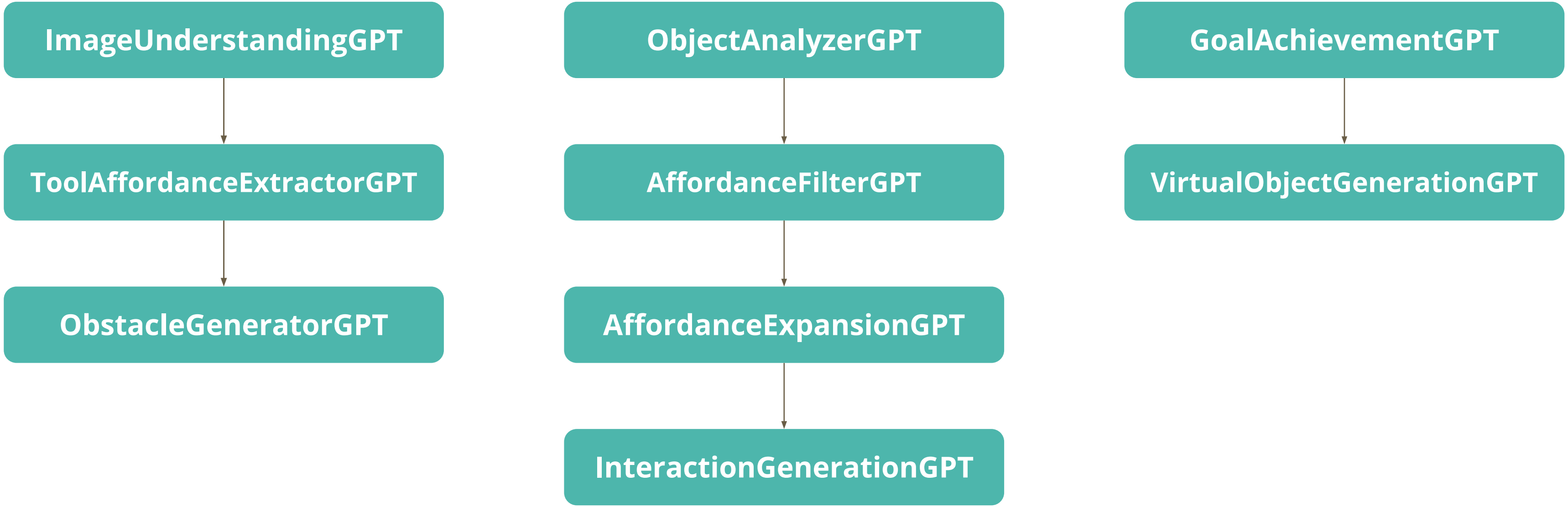
```python
class NodeObstacle(BaseModel):
    reasoning: str
    SELECTED_NODE: str
    OBSTACLE_DESCRIPTION: str
    OBSTACLE_INITIAL_STATE: str
    OBSTACLE_FINAL_STATE: str
    TOOL_OBJECT_INDEX: int
    PRIMITIVE_ACTION_TYPE: str

response_format = NodeObstacle
node_obstacle = call_openai(prompt, response_format, client)
```

```json
"type": "object",
"description": "The address object to insert into the database",
"properties": {
    "number": {
        "type": "string",
        "description": "The number of the address. Eg. for 123 main st, this would be 123"
    },
    "street": {
        "type": "string",
        "description": "The street name. Eg. for 123 main st, this would be main st"
    },
    "city": {
        "type": "string",
        "description": "The city of the address"
    }
},
"additionalProperties": false,
"required": [
    "number",
    "street",
    "city"
]
```

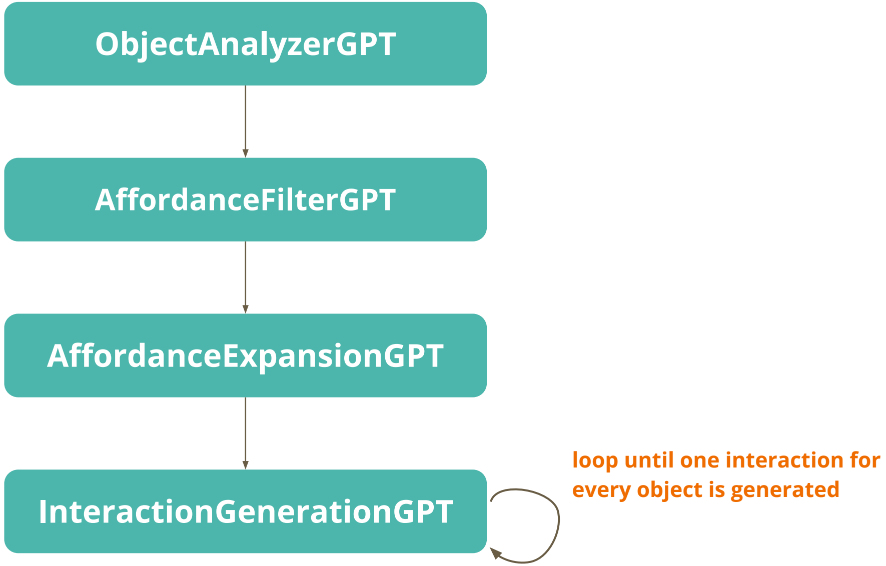
Integrating LLM calls into code

Module chaining



- We can do everything in one prompt, but it will have to do many different tasks at the same time.
- If we define prompts as steps of a system and keep the functionality of each module well-scoped, we can generate richer results.

Module chaining



These modules can be reused or called in a loop until a specific condition is met (for example, if all real objects are used)

Traditional code

- How do we create graphs or steps in the puzzle?
- How do we integrate the different modules?

Do we let the LLM generate graphs together directly and connect modules directly?

→

We don't need to.

Traditional code is still way more reliable.

We only need LLMs for creative content generation.

Graph networks in Python

- On your terminal run:
    - *pip install networkx*
- To import the NetworkX library, use:

```
import networkx as nx
```

- [GraphX Documentation]()

```
G.add_edge(1, 2)
H = nx.DiGraph(G)  # create a DiGraph using the connections from G
list(H.edges())
```

```
[(1, 2), (2, 1)]
```

```
edgelist = [(0, 1), (1, 2), (2, 3)]
H = nx.Graph(edgelist)  # create a graph from an edge list
list(H.edges())
```

```
[(0, 1), (1, 2), (2, 3)]
```

Example generated interaction

```
}
{
  "TOOL_OBJECT": "plunger",
  "NEW_INITIAL_STATE": "flattened",
  "NEW_FINAL_STATE": "shaped",
  "STATE_CHANGE_REASONING": "The dough initially needs to be flattened and shaped before it can be baked. Using the plunger, the dough is pressed and molded into the desired shape, t
ransitioning it from a 'flattened' state to a 'shaped' state. This shaping process is crucial as it prepares the dough for the baking process, ensuring it cooks evenly and achieves t
he desired texture and form. Once the dough is shaped, it can then proceed to the baking stage, transitioning from 'unbaked' to 'baked'."
}

Nodes in graph:
[('dough: unbaked → baked', {'type': 'state_change'}), ('dough: flattened → shaped', {})]

Edges in graph:
[('dough: flattened → shaped', 'dough: unbaked → baked', {})]
```

How to build the pipeline?

- Do we start at the goal and let the LLM generate what objects can solve this goal and find ways all objects can be integrated into the puzzle?
- Or do we start by looking at the objects, extracting their affordances and generating interesting interactions between them that get us closer to the goal?

Goal: Kill the angry dragon

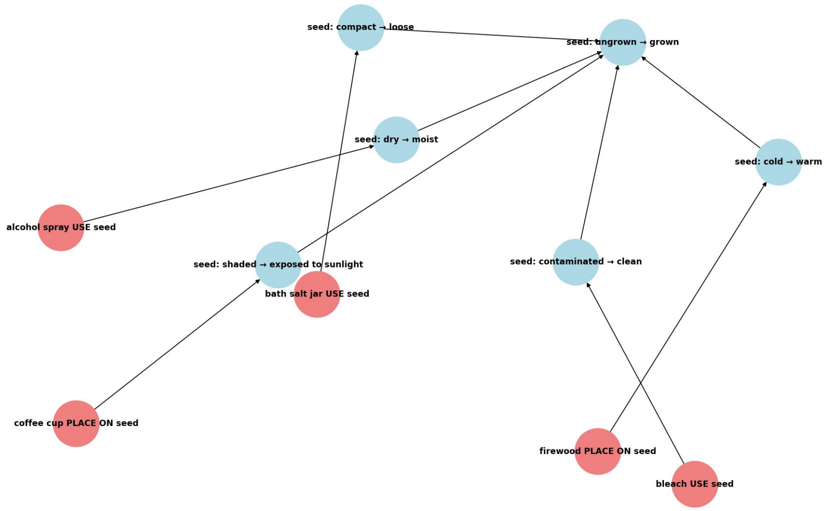
This is one of the design questions: whether we want a top-down, bottom-up or mixed approach

To Do

Basic

Generate a set of puzzle graphs

- Write the pipeline to generate puzzles from goals and sets of real objects
- Think about your own goal set with 3 different goals
- First:
    - Use the following set of 5 real objects:
        - bucket, hammer, coffee cup, alcohol spray, umbrella
    - Combine each of the goals with the set of real objects to generate 3 different puzzles
- Then:
    - Come up with a new object set by yourself
    - Repeat the same process with the same goal set to generate 3 other puzzles
- **Puzzles must use all objects in the set in some way or another**



Report

- Summarize what you did in this lab.
  - Show the basic and bonus you implemented.
  - Use screenshots & video demos to show your results.
  - Make sure you show both the generated interactions that are actually used in the puzzle graph and the natural-language story/description behind each interaction.
- What you did to improve this pipeline. Or how you can improve this pipeline.
- Anything related to this lab.

Additional

- Come up with edge cases for the sets of objects and goals
  - Are there any weird interactions? Is there a pattern in what kind of objects or goals lead to weird interactions?
- Try GPT's vision capabilities to obtain object definitions from the real world instead of hard-coding them
  - From a picture and a prompt designed by you, you can obtain the definitions in the format that you need
- Connect the puzzle graphs to Unity
  - How can the puzzle graph be used in actual interactive experiences? Data can be read in Unity code to define the flow of the Unity game.