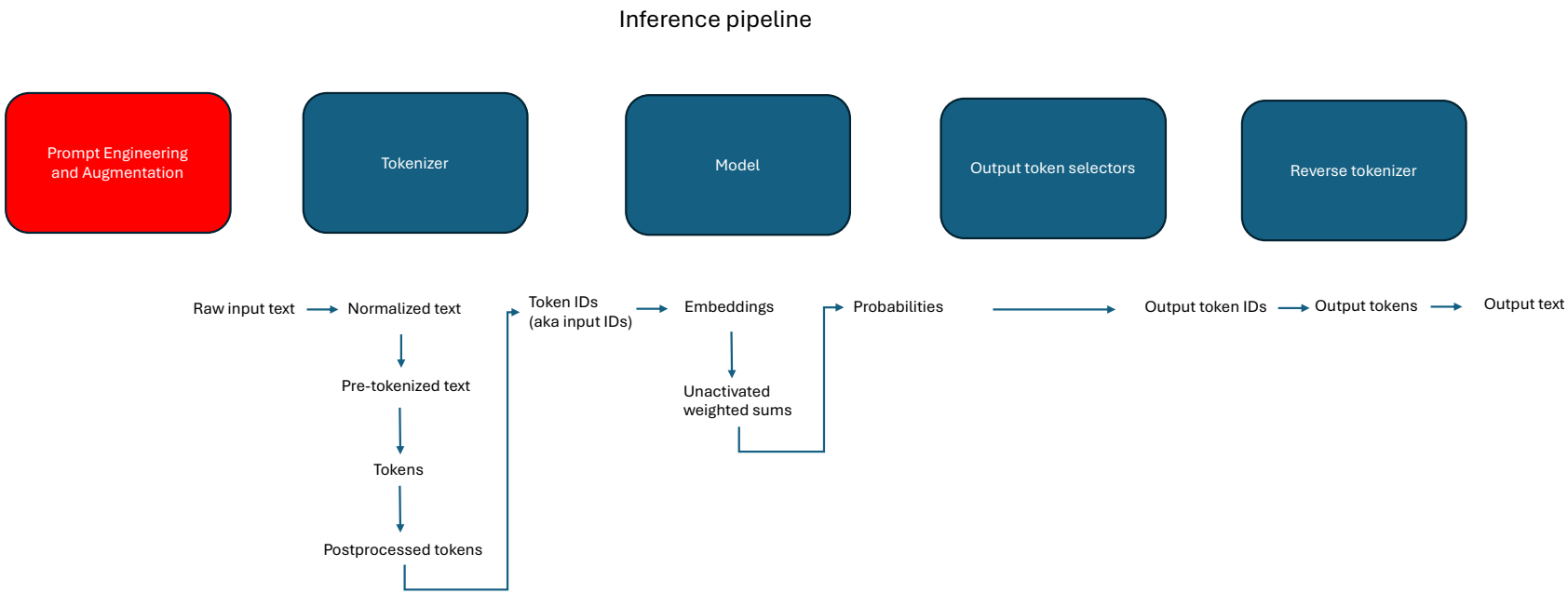
BZAN 583:
Generative AI with Large Language Models for Business Applications

Prompt Engineering and Prompt Augmentation

Michel Ballings

Outline I

Inference Pipeline Overview



Outline I

Outline I

Basic prompt

- The most basic of prompts is a sequence of words without a question or an instruction for the model to do anything. For example: 'We love'
- The model then predicts what should come next and completes the sentence. The generated text is called a completion. For example the model may complete the sentence 'We love', with 'cookies'.
- We could say that the instruction to complete the text is an implicit instruction. Put differently, we can say that we are using the default instruction.
- LLMs can do so much more than simply completing text. To tap this power we need a more structured prompt

Outline I

Instruction-based prompting

- With instruction-based prompting we give the LLM an explicit instruction: answer a specific question, complete a specific task, ...
- For example: 'Summarize the following text. Text: {text to summarize}. Summary:'
- The words 'Text:' and 'Summary:' are indicators. Even if the model has not been trained on these components directly, it has seen enough instructions to generalize to this prompt structure.
- Common prompt components:
  - Instruction: summarize, classify, answer, explain, mask personally identifiable information, write, ...
  - Tone: 'Write in a formal tone'
  - Response format: 'Answer the question in ten words or less.', 'Answer in exactly five bullet points.', 'List the top three concerns.'
  - Persona: 'You are an expert in statistics. Explain ...'
  - Context: 'I am asking the following question because ...'
  - Data: Additional data required for the task.
  - Audience: 'Answer the question as if you are speaking to a layperson.'

Outline I

Context window length

- The ideal prompt structure depends on the task as well as the model's context window.
- The context window refers to the numbers of tokens the model can take as input. 200,000 tokens is approximately 500 pages of text.
  - FLAN-tT5 has a 512 token long context window.
  - Meta's Llama 4 has 10 million tokens (Llama 2:4096, Llama 3: 128,000),
  - Anthropic's Claude has 1 million tokens (Claude 2: 100,000, Claude 3: 200,000)
  - Google's Gemini 3 has 1 million tokens
  - OpenAI's ChatGPT 4 has 128,000 tokens
- Tokens beyond the context length will be ignored.

Outline I

Outline I

N-shot inference

- In the previous section we described what the LLM should do.
- Here we go one step further and we provide examples to the LLM.
- This is called **in-context** learning.
- Zero-shot inference means we do not provide examples.
- One-shot inference means we provide one example.
- Few-shot inference means we provide at least two examples.
- Example of one-shot inference: 'Classify the following text as negative, neutral, or positive. Text: This is a great cookie. Sentiment: positive. Text: The cookie was OK. Sentiment:'.

Outline I

In-context learning gone wrong

Example of one-shot inference that can throw off the model: ‘Classify the following text as negative, neutral, or positive. Text: This is a great cookie. Sentiment: negative. Text: The cookie was bad. Sentiment:’.

Outline I

In-context learning best practices

- Start with zero-shot prompting and work your way up to few shot inference
- If we are not getting results with five examples we may need to try a different model.
- Consider the context window length. With multiple examples we can quickly surpass the window length.
- See **incontextlearning.py**

Outline I

Chain prompting

- Chain prompting is a way to break up the problem.
- The technique involves breaking up the prompt in multiple smaller sequential prompts. This can often improve the performance of LLMs because it simplifies the tasks for the LLM and allows the LLM to spend more time on each individual question instead of responding to the whole problem.
- In chain prompting the output of previous calls to the LLM is part of the input to next calls.
- For example:
  - Input: Generate a product name for a chocolate cookie bar. Output: Snickers.
  - Input: Generate a list of ingredients for chocolate cookie bars called Snickers. Output: caramel, chocolate, sugar.
  - Input: Generate reasons why we should eat chocolate cookie bars called Snickers with ingredients caramel, chocolate, and sugar . Output: It is the most popular cookie bar in the world.

Outline I

Outline I

Chain-of-thought

- Chain-of-Thought (Cot) prompting encourages the model to think (and share its thoughts) before answering
- Example (google/flan-t5-xl):

	Zero-shot standard prompt	Zero-shot chain-of-thought prompt
Prompt	Solve: How many cookies are left if you start with 10 cookies, give away 3, then buy 5 more.	Solve: How many cookies are left if you start with 10 cookies, give away 3, then buy 5 more. **Think step-by-step.**
Answer	7	After giving away 3 cookies, you have 10 - 3 = 7 cookies. After buying 5 more cookies, you have 7 + 5 = 12 cookies. The answer: 12.

- We can also use one or few shot inference and provide examples of how to solve problems step by step.
- See *chain_of_thought.py*

Outline I

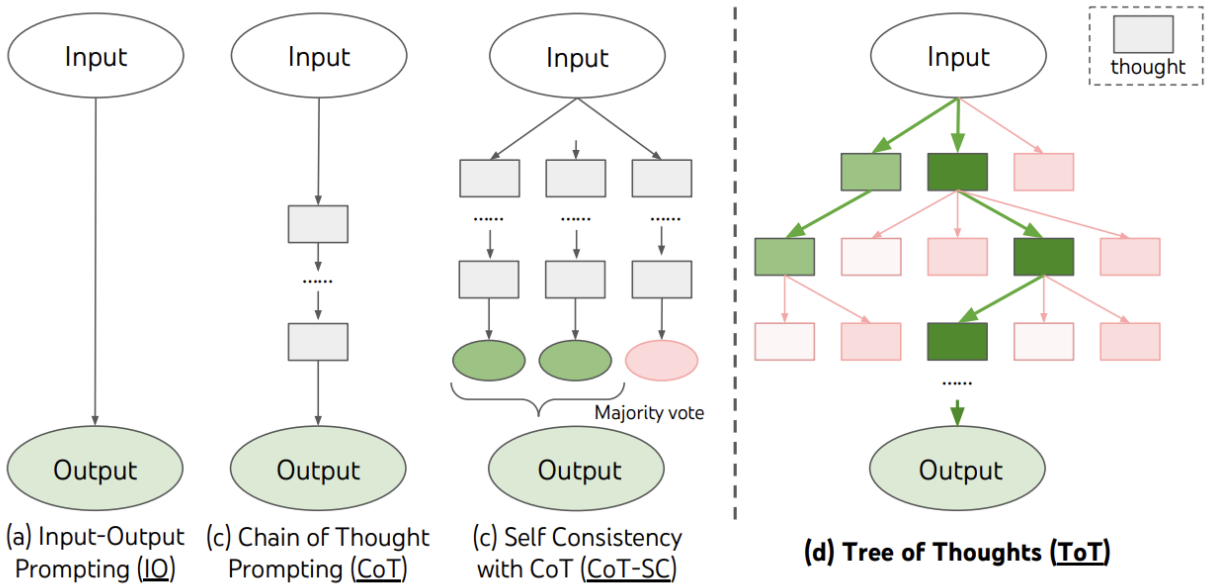
Self-consistency

- When we are using sampling, the same query can lead to different results.
- For example, if we run the model ten times with the prompt "Solve: How many cookies are left if you start with 10 cookies, give away 3, then buy 5 more. Think step-by-step.", and top-k sampling, we may receive different answers.
- For example, the ten runs may produce {10:1,11:3,12:4,13:1,14:1} where the number before the colon is the answer, and the number after the column is the frequency of occurrence.
- To determine the final answer we can simply pick the answer with the most votes, here the mode: 12.
- This idea is similar to ensembling multiple models.

Outline I

Tree-of-thought

- Tree-of-thought promotes deep exploration of several ideas. The Figure below comes from `https://arxiv.org/pdf/2305.10601` and compares the main methods we have seen.



- This method delivers performance improvements by considering multiple paths of thinking.
- The main disadvantage is that it requires many calls to the LLM and is thus very expensive and slows down the text generation process.

Tree-of-thought

- A solution to convert the tree-of-thought setup into a simple prompt has been presented in `https://github.com/dave1010/tree-of-thought-prompting`
- Prompt: "Imagine three different experts are answering this question. All experts will write down one step of their thinking, then share it with the group. Then all experts will go on to the next step, etc. If any expert realizes they're wrong at any point then they leave. The question is..."
- We can then expect a response per the following format (here the question was about a ball and where it went):

  Expert 1: The ball is in the living room.

  Expert 2: The ball is in the kitchen.

  Expert 3: The ball is in the bedroom.

  Expert 1: Bob carries the cup to the bedroom, so the ball must be in the cup.

  Expert 2: Oh, I see my mistake. Yes, the ball is in the cup.

  Expert 3: Agreed, the ball is in the cup in the bedroom.

  ...

  All three experts agree that the ball is in the bedroom.
- See *tree_of_thought.py*

Outline I

Output verification

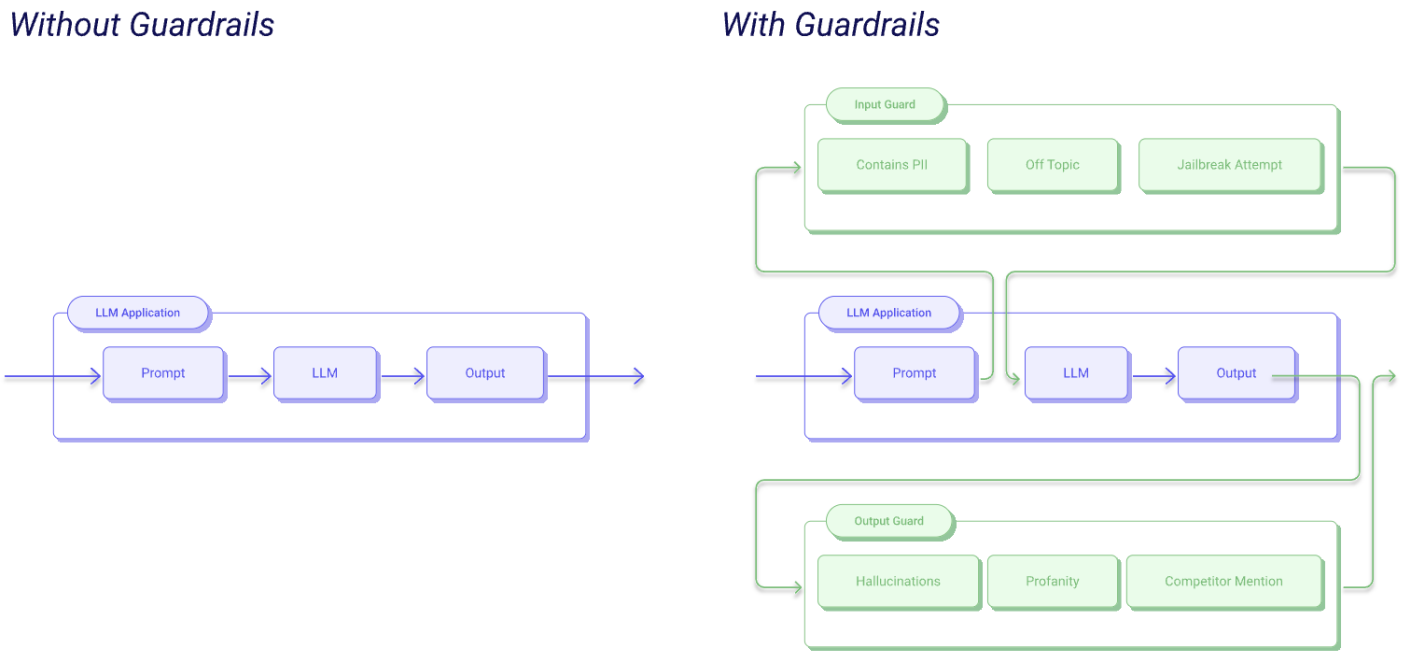
- When systems are used in production, guardrails are important to generate an experience that is consistent with expectations.
- Guardrails can include validating the format, accuracy, and bias.
- In the introduction we mentioned three ways to influence the model using filtering: changing the prompt, changing the training data (e.g., for finetuning), filtering the output.
- Output filtering example. If we know that the model should only return True or False, we could filter out all the other tokens during next token selection and allow it to only select among True and False.
- Input filtering example. If we know that we want the result in a certain format, we can use in-context learning or provide instructions regarding the format.

Output verification

- JSON prompt example:
  """Invent a new cookie and then describe it using the following format:
  {
  "name": "<the cookie's name>",
  "recipe": "<the cookie's recipe>",
  "taste": "<the cookie's taste>"
  } """

Output verification

- Creating guardrails is a lot of hard work and developing high-quality systems goes beyond keyword search and involves creating datasets for training models that can detect guardrail violations.
- Here is one example of a guardrail package (`https://github.com/guardrails-ai/guardrails`) that is attracting



- See *output_verification.py*

Outline I

Chains

- Chaining and chains refers to connecting methods and modules
- So far, we have seen one example of chaining; chain prompting.
- The *langchain* module provides methods for everything related to chaining.
- The more recent module *langchain_huggingface* provides an easy interface between langchain and Huggingface methods.
- We will see code examples for
  - a basic chain connecting a prompt template and a model, and
  - a reimplementation of our chain prompting example that we saw earlier.
- On the next slide we have a look at the chain prompting example as preparation for the implementation.

Chain prompting example

In one of the bonus assignments we had an example of chain prompting. The goal was to first create a brand name for chocolate chip cookies, then create a list of ingredients, and then elaborate on how the proposed chocolate chip cookies are different from other chocolate chip cookies. Subsequent calls to our LLM can be summarized as follows:

- brand name = f(product)
- ingredients = f(product, brand name)
- difference = f(product, brand name, ingredients)

See *chain.py*

Outline I

Memory

- LLMs will not remember any inputs from before; they are said to be stateless.
- To make them stateful and give them a memory of what happened before, we can create a wrapper around the LLM that will store previous inputs. Here are three variants.
  - Conversation buffer: append new prompts to previous prompts.
    - Advantages: no information loss.
    - Disadvantages: slower generation speed, will not work with small context windows
  - Windowed conversation buffer: retain only the last k inputs by appending new prompts to previous prompts, and deleting the oldest ones.
    - Advantages: works with small context windows
    - Disadvantages: information loss, because the LLM will only remember the k most recent inputs
  - Conversation summary: use another call to the LLM (may be another LLM) to summarize the chat history at any given point of interaction.
    - Advantages: captures full history, can work with small context windows
    - Disadvantages: an additional call is required for each interaction, the quality depends on the LLM's summarization capabilities.
- See *memory.py*

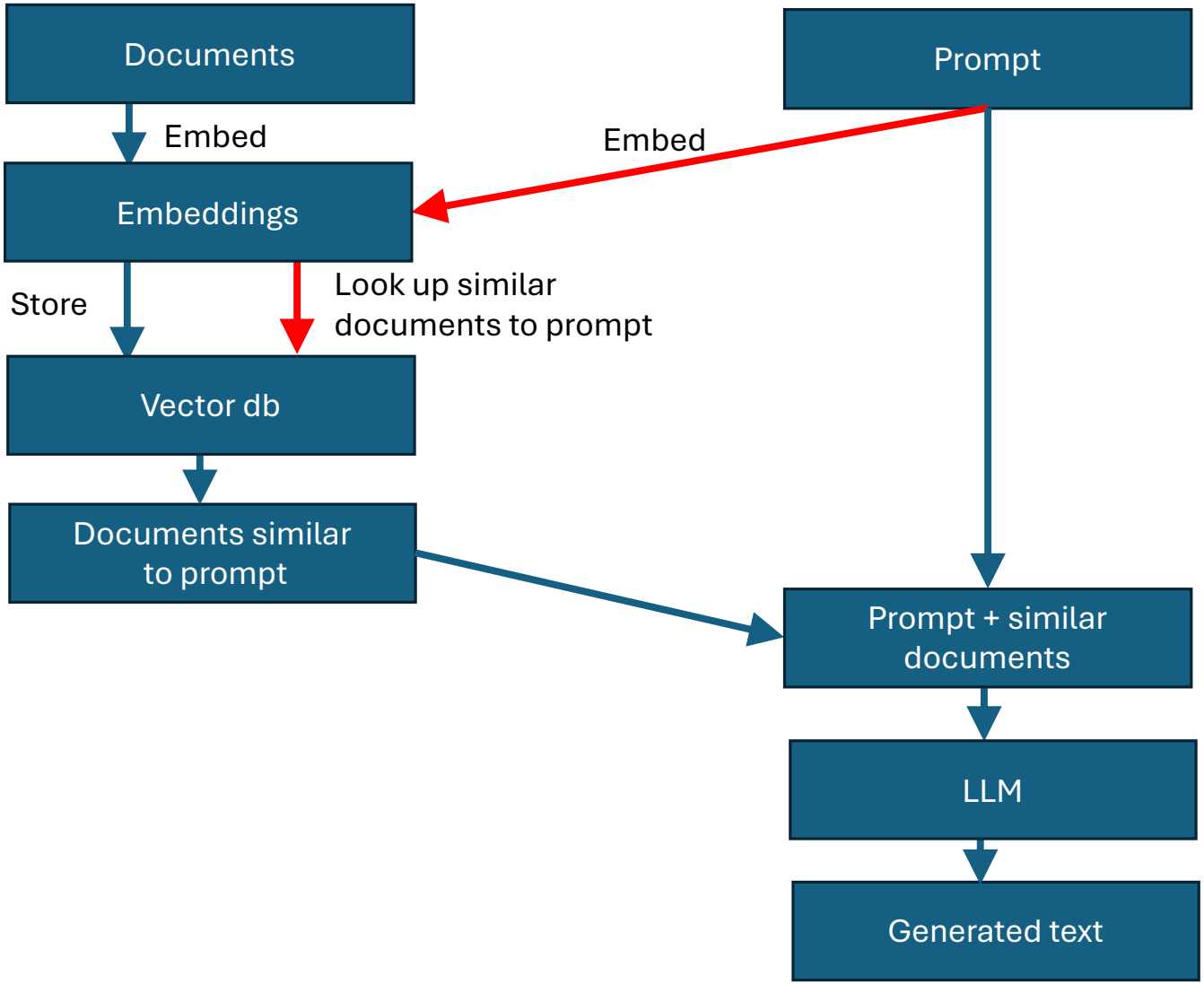
Outline I

Prompt engineering best practices

- Be specific in what you want. Instead of ‘Summarize this paragraph’, we can ask ’Summarize this paragraph in two sentences or less.’
- Tell the LLM that it should let you know when it is not sure about its response. ‘Answer I don’t know if you are unsure’. This will help with avoiding hallucinations.
- Use the appropriate order. Put the instruction either at the beginning or end of the prompt as information in the middle is often de-emphasized.

Outline I

Retrieval Augmented Generation



Retrieval Augmented Generation

- Retrieval augmented generation (RAG) (`https://arxiv.org/abs/2005.11401`) incorporates search into the generation pipleine.
- Reduces hallucinations and uses what is called **semantic search**, which denotes searching by meaning as opposed to searching by keyword matching.
- High-level overview of RAG:
  - Step 1: embed all documents and store them in a vector store (aka vector database, embeddings database)
  - Step 2 (**dense retrieval**): embed the prompt, and then retrieve the k nearest neighbor documents of the embedded prompt from a set of embedded documents.
  - Step 3: (**reranking**, not shown in diagram on previous slide): a separate model, called the reranker or reranker model, takes as input the embedded nearest neighbor documents, and the embedded prompt, and then scores each of the documents for relevance to the prompt, and orders them per that score. We can then keep the top documents or keep them all.
  - Step 4: (**grounded generation**): use the prompt and the (reranked) documents as input to an LLM for text generation.

Retrieval Augmented Generation

Notes:

- **Chunking.** The documents that we have in step 1, will eventually be used as input for the LLM and each LLM has a limited context length. Therefore the documents need to be chunked and so we will end up with multiple vectors per document. Chunking process:
  - Choose how many tokens each chunk should contain.
  - Split up the document in chunks, and add some of the tokens before each chunk and some of the tokens after each chunk.
  - There will be some overlap between chunks but by including some surrounding text that also appears in adjacent chunks we can capture a great deal of context.

  The above process is more general than chunking by sentence (too little context) or paragraph (only works if paragraphs are short enough for the context window).
- In step 2 (KNN), note that we are retrieving the nearest neighbors (the most similar document to the prompt), but even the most similar documents may be very dissimilar. Therefore, it makes sense to use a minimum similarity threshold to decide if the nearest neighbors should be returned. This can also be done in step 3 (reranking).
- *RAG.py*

Outline I

Agents: Program-aided LLMs

Agents are chains. We teach the LLM to generate code or specific keywords. We then parse their entire generated output and search for that code or keyword. If we find it we execute the code.
Example:

- We teach the model with in-context learning to output calculator(1+1) when it encounters 'What is 1+1?'.
- We then parse the generated text for the presence of 'calculator('.
- If it is present, we extract everything in between the parentheses: '1+1'
- We then evaluate the string '1+1' in python.

See agents.py