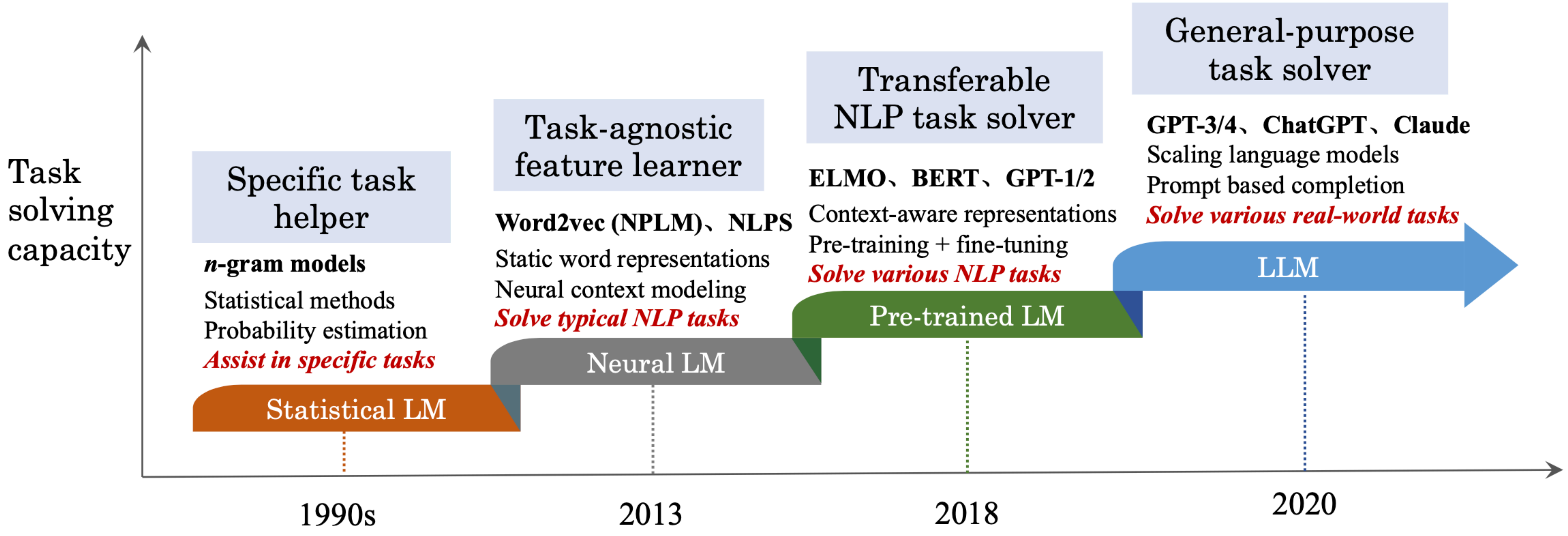
LLM & Agent Basics

Contents

What Makes an LM an LLM

- Language Modeling: predicting the probabilities of future (or missing) tokens [1]

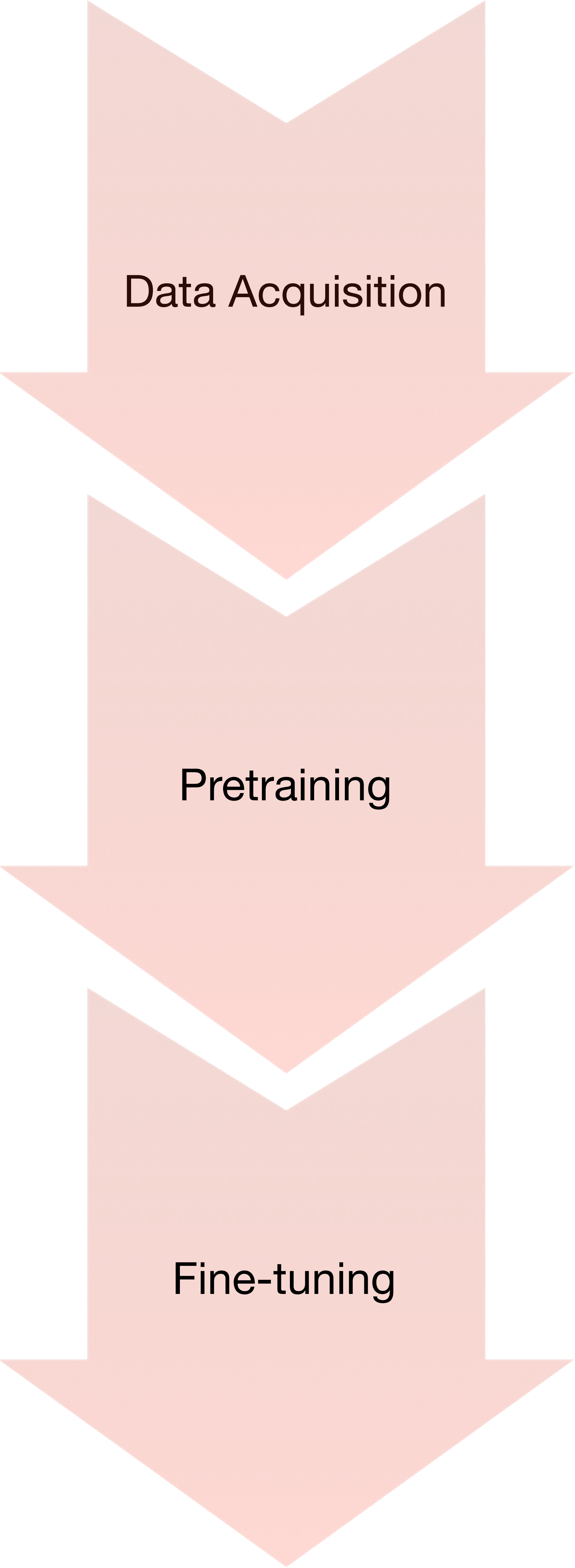


- Scaling of LMs (mostly model size and data size) increases the capacity of the model on downstream tasks
- A language models becomes a Large Language Model when it reaches a certain **scale**. Reason: having crossed a certain threshold, the scaled LMs exhibit abilities unavailable on smaller models: *emergent abilities* [1, 2]. Usually, LMs with at least hundreds of millions parameters (mostly billions) have emergent abilities

What Makes an LM an LLM

- Scaling laws estimate the performance of larger models based on that of smaller models; usually, model size, data size, and computational budget are taken into consideration [1]
  - General principle: the bigger the scale, the better the result („Scale is all you need“)
  - Allows to inspect the impact of different techniques and tricks on smaller models while understanding the expected impact at scale [1]
  - Inverse scaling: performance on some tasks can decrease with scaling [1]
- Emergent abilities do not follow the scaling laws and appear mysteriously after a certain scale was achieved. They do not exist in small models but suddenly appear in large ones, beyond what can be predicted from simple the scaling laws [1, 2]
  - No universal threshold exist
  - It is challenging to predict if and when these abilities emerge
  - Examples: in-context learning, step-by-step reasoning, instructions following [1, 2]
- => Both the scaling laws and the phenomenon of emergent abilities suggest that scaling is beneficial
- Many tasks can be formulated as a next word prediction problem => paradigm shift: LLMs pose complex **task solving** as their mainly goal, not just pure language modeling [1]

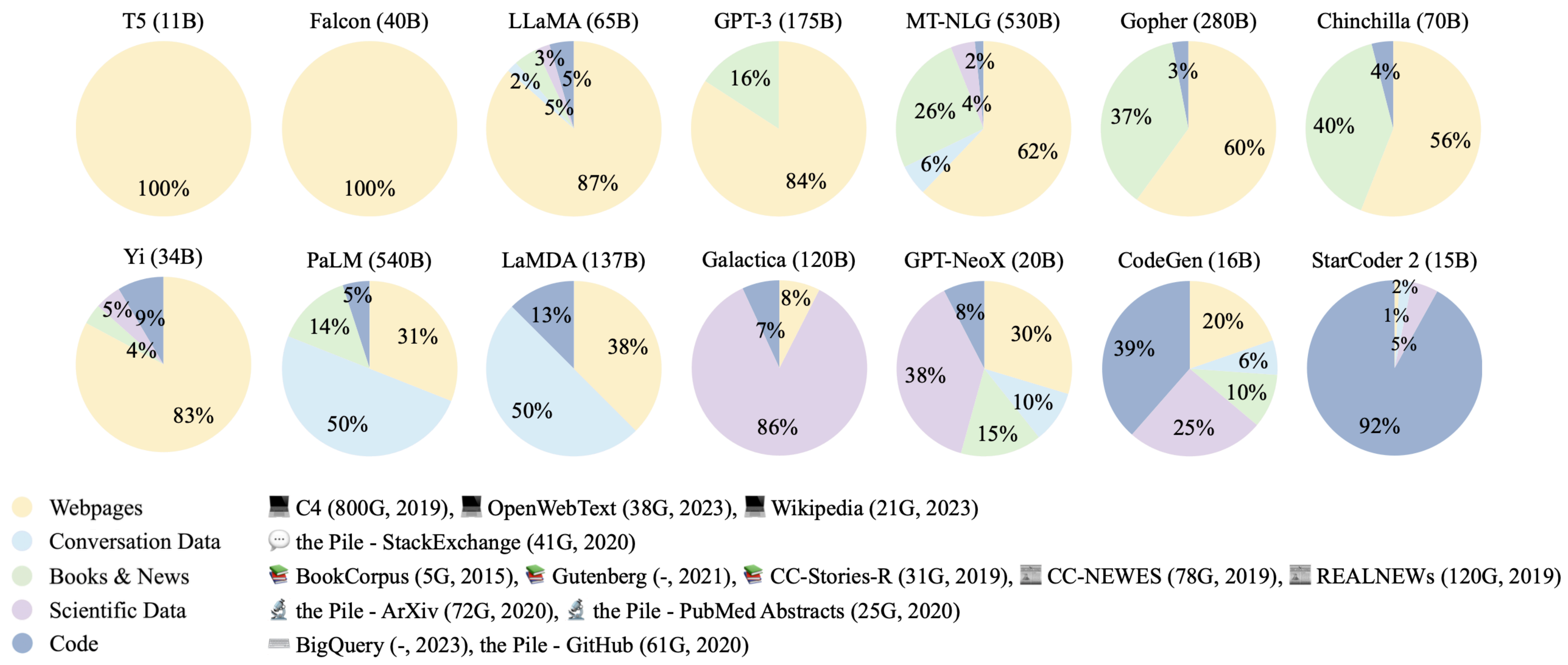
How to Build an LLM



- Collect and preprocess the data for pretraining and fine-tuning
- Unsupervised learning on a vast amount of general-purpose data. Pretraining establishes the basis of the abilities of an LLM
- Supervised learning on a smaller amount of task-specific data

How to Build an LLM

- Misture of diverse open-source textual data is used as a pretraining corpus: webpages, dialogs, books, code
- The data is cleaned, duplicates and sensitive information are removed
- Quality and diversity as well as mixture proportions of data influence the performance of the model [1]; important principle: „Garbage in — garbage out“
- Data curriculum: ordering different parts of pretraining data in a certain sequence might improve the performance (e.g. more sophisticated tasks follow after more general-purpose ones) [1]



How to Build an LLM

- Architecture: transformers (mostly decoder-only) with different types of attention
- Much engineering is involved when pretraining (and fine-tuning):
  - Hyperparameter tuning
  - Improving computer efficiency: parallelization, mixed precision training
- The most frequent training objective is language modeling [1]:

$$\mathcal{L}_{LM}(\mathbf{x}) = \sum_{i=1}^{n} \log P(x_i|\mathbf{x}_{<i}).$$

“Given a sequence of tokens **x**, predict log probabilities of each token x in the vocab to come next“

- Sometimes, denoising autoencoding is utilized [1]:

$$\mathcal{L}_{DAE}(\mathbf{x}) = \log P(\tilde{\mathbf{x}}|\mathbf{x}_{\backslash \tilde{\mathbf{x}}}).$$

“Given a sequence of tokens **x** where its subsample $\tilde{\mathbf{x}}$ was corrupted, predict log probabilities of different $\tilde{\mathbf{x}}$’s“

- LLMs may employ different decoding strategies:
  - Greedy decoding takes the most probable next token
  - Random sampling takes a random token based on their probabilities
  - There might be constrains to the minimal probabilities for random sampling

How to Build an LLM

- The goal of fine-tuning is to improve/unlock certain domain-specific abilities of LLMs beyond the general-purpose capabilities they obtain after pretraining
- Usually, a much smaller amounts of high-quality labelled data is taken for fine-tuning [1] (cf. ~300B tokens for pretraining and ~50k examples for fine-tuning for InstructGPT)
- There are two main goals that are set for fine-tuning: *instruction tuning* and *alignment tuning*
- Instruction tuning utilizes example pairs of given instructions and desired output for different kinds of tasks (giving a recommendation, solving a problem, generating a JSON, multi-step reasoning etc.)
- Alignment tuning is used to align the LLM with human preferences (being honest, helpful, harmless)
  - Reinforcement learning is widely used (e.g. RLHF — reinforcement learning with human feedback)
  - Non-RL methods are similar to instruction tuning, but instructions persuade the goal of alignment [1]

Prompting

- There are 4 key ingredients to a prompt [1]:
  - Task description: what you want to achieve
  - Input data: the input data for your task
  - Context: anything that is needed for solving the task
  - Style: how you describe the task (description should be unambiguous, decomposition onto subtasks is usually helpful etc.) [1]
- Since most of the data for LLMs is English, using English instructions yields a better result even when working with non-English input data [1]
- It is widely shown that providing demonstration and utilizing chain-of-thought prompting boosts the performance of LLMs [1, 3, 4]

TABLE 13: Example instructions collected from [447, 457]. The blue text denotes the task description, the red text denotes the contextual information, the green text denotes the demonstrations, and the gold text denotes the prompt style.

Use the provided articles delimited by triple quotes to answer questions. If the answer cannot be found in the articles, write "I could not find an answer."

Articles: """Joao Moutinho is a Portuguese footballer who last played as a central midfielder for Premier League club Wolverhampton Wanderers and the Portugal national team."""

Question: Is the following sentence plausible? 'Joao Moutinho was out at third.'

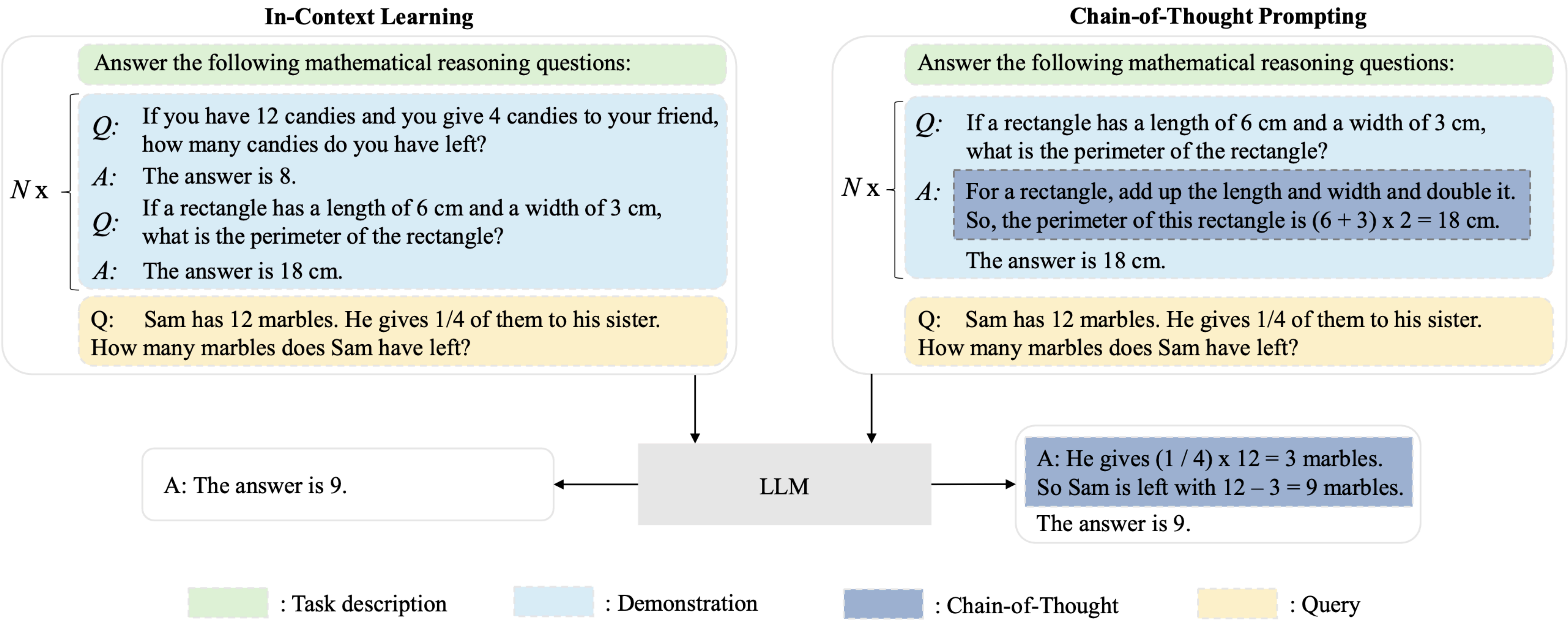
Answer: Let's think step by step. Joao Moutinho is a soccer player. Being out at third is part of baseball, not soccer. So the answer is No.

...

<Demonstrations>

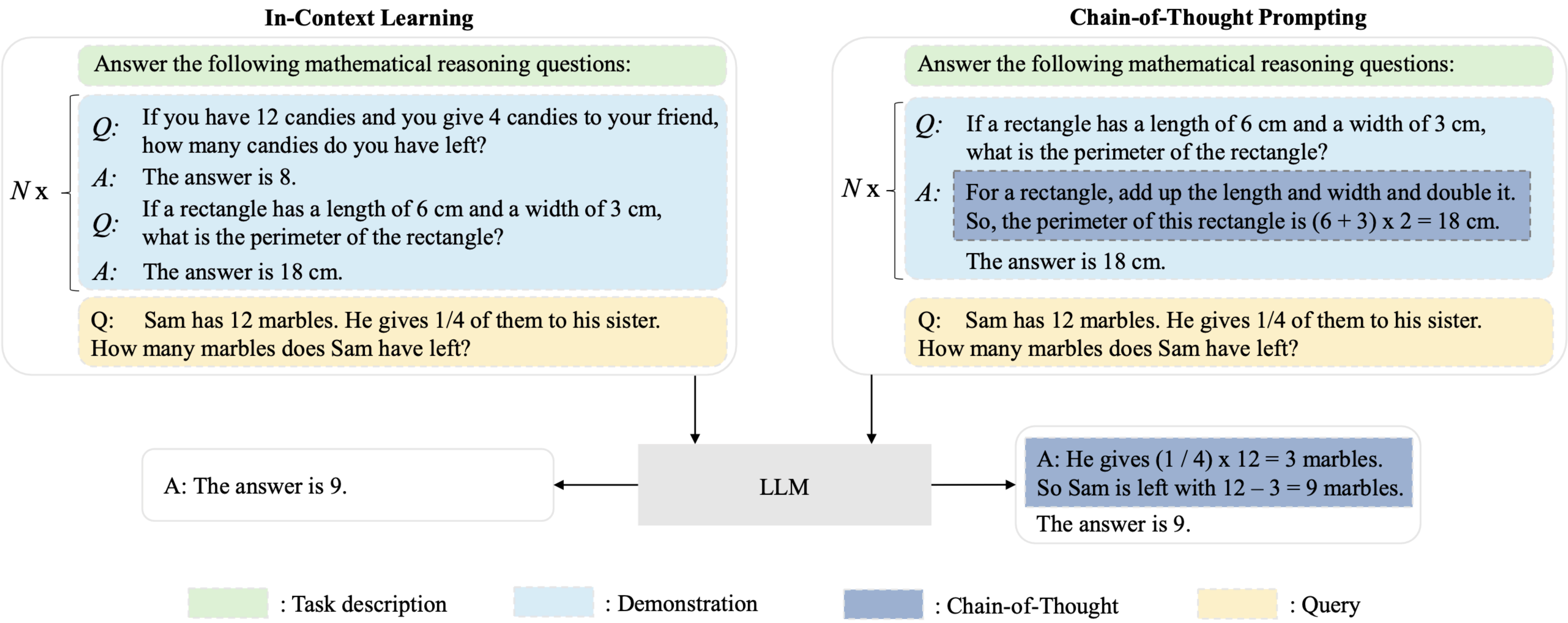
In-context Learning

- In-context learning (ICL) is the ability of an LLM to handle a task it has never seen by utilizing examples of how this tasks is solved
- ICL is an *emergent ability*
- Few-shot prompting stimulates ICL in LLMs
- Ordering of demonstrations is important because of the *recency bias* — the effect of repeating the last example [1]
- It is not recommended to use demonstrations of how the actual input data should be dealt with



Chain-of-Thought Prompting

- Chain-of-Though (CoT) prompting is an extension of ICL
- Instead of `<input, output>` demonstrations, CoT prompting suggests to give demonstrations of form `<input, CoT, output>`
- This brings a noticeable performance gains on various tasks [6]
- Opinion: ICL and CoT reasoning arise as a result of consuming of a large volume of code



Structured Output

- „We need structured output“: *structured output* is essential when integrating LLMs into real-world pipelines [3]. To efficiently use the generated data in various pipelines, it is essential to have certain structure constraints (JSON payloads in web-development, rendering content on the frontend etc.)
- Structured output generation is an ability that is mostly taught to LLMs on during fine-tuning [7]
- Modern LLMs are capable to **ensure** the desired output format [8]
- Structured output is also essential when making LLM-based pipelines because it allows for reliable transfer or (generated) knowledge between the agents
- Overall, using structured output reduces cost and effort of development of LLM-based applications [3]

Tool Calling

- The knowledge of the LLMs is never up-to-date
- LLMs struggle to perform some symbolical tasks (arithmetics, counting etc.)
- Most of the real-world tasks require real-time information search, performing actions, interaction with the environment etc.
- => It is extremely beneficial to connect external tools to LLMs
- Solution: *tool calling*. Tool calling is basically the same thing as generating a structure output, but instead of generating answer to the query, input parameters for one of the predefined tools is generated
- Structured descriptions of tools are provided to the model, and it decides if it should utilize them, and if yes, which one and with what parameters

```
{
  "name": "get_weather",
  "description": "Fetch weather
info for a city.",
  "parameters": {
    "city": "string",
    "unit": "string"
  }
}
```

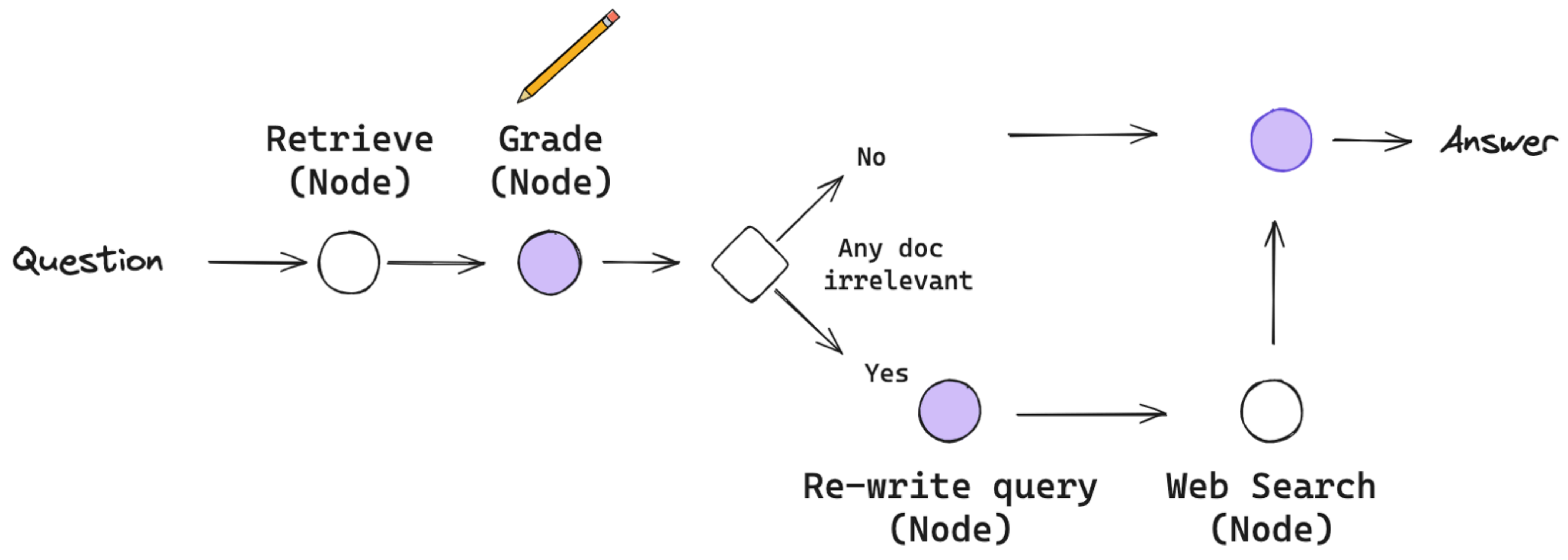
```
User: What's the weather in Tokyo?

Assistant: {
  "tool": „get_weather",
  "parameters": {
    "city": „Tokyo",
    "unit": „celsius"
  }
}
```

- Just as generating structured output, tool calling is taught to the model during fine-tuning [1, 8], mostly based on open-source APIs [9, 10]
  - Some approaches mix tool calling data into the pretraining data [11]

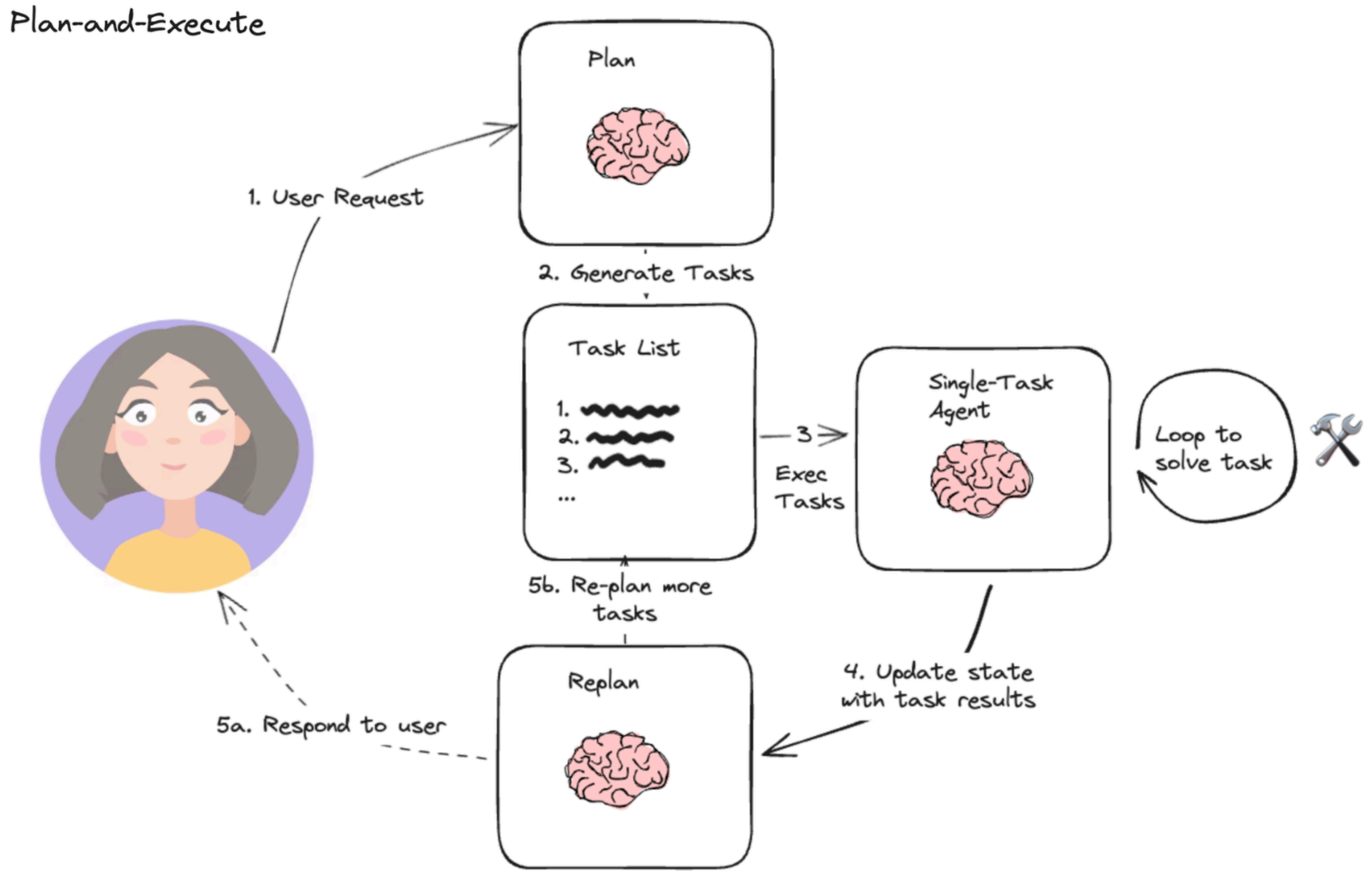
Piping & Planning

- Complex tasks cannot be solved in a simple input-output manner
- For complicated tasks, *piping* is used: different LLMs and tools are connected into a pipeline so that the output of one LLM/tool becomes the input of the next one
- Most often, complicated conditional piping is required; a prominent approach is to create *graphs* that would have LLMs and tools as nodes and rules of transition as *edges*
- That allows for decomposing the tasks into smaller subtasks that are handled separately by the nodes designed specifically for this subtasks



Piping & Planning

- One of the prominent pipeline designs is *plan-and-execute* [1, 13]
- In such a pipeline, the *task planner* generates a plan, *plan executor*(s) execute it, and the *environment* gives signals about the executions
- Many of the pipelines utilize *reflexion*, when a separate agent reflects on the results obtained so far and sends a signal if the pipeline should process or if there are refinements necessary
- Thus, a self-improvement loop is created which can improve the resulting performance



References

[1] A Survey of Large Language Models, Renmin University of China et al.

[2] Emergent Abilities of Large Language Models, Google Research, Stanford, UNC Chapel Hill, DeepMind

[3] “We Need Structured Output”: Towards User-centered Constraints on Large Language Model Output, Google Research & Google

[4] Agent Instructs Large Language Models to be General Zero-Shot Reasoners, Washington University & UC Berkeley

[5] Language Models are Few-Shot Learners, OpenAI

[6] Chain-of-Thought Prompting Elicits Reasoning in Large Language Model, Google Research

[7] The Llama 3 Herd of Models, Meta AI

[8] Introducing Structured Outputs in the API, OpenAI

[9] Tool Learning with Large Language Models: A Survey, Renmin University of China et al.

[10] ToolACE: Winning the Points of LLM Function Calling, Huawei Noah’s Ark Lab et al.

[11] Toolformer: Language Models Can Teach Themselves to Use Tools, Meta AI

[12] Granite-Function Calling Model: Introducing Function Calling Abilities via Multi-task Learning of Granular Tasks, IBM Research

[13] Berkeley Function-Calling Leaderboard, UC Berkeley (leaderboard)

[14] ReAct: Synergizing Reasoning and Acting in Language Models, Princeton University & Google Research

[15] A Survey on Multimodal Large Language Models, University of Science and Technology of China & Tencent YouTu Lab