Prompt chaining, routing, and planning

Shreya Havaldar

Announcements

- My OH will be 1-2 pm on Wednesday (tomorrow) this week!
- Sign up for a HW 3 demo slot if you haven’t already!

Why are single prompts not enough?

Even with instruction tuning, LLMs struggle with complexity:

- They are fundamentally next-token predictors.
- They do not inherently manage control flow.
- They struggle with long-horizon tasks.
- They lack built-in error correction.

We must design structure around them!

Single Prompts: Failure Modes

Example task:

> *“Research competitors, extract pricing models, compare positioning, generate strategy memo.”*

Failure modes: Hallucinated data, missed constraints, incorrect structure, shallow reasoning

→ **Too much cognitive load is forced into one generation.**

Single Prompts: Failure Modes

Example task:

> *“Research competitors, extract pricing models, compare positioning, generate strategy memo.”*

Failure modes: Hallucinated data, missed constraints, incorrect structure, shallow reasoning

→ **Too much cognitive load is forced into one generation.**

Long-Horizon Tasks

Long-horizon tasks require extended sequences of actions, typically involving more than 10 individual steps, or over a prolonged period (e.g., several minutes) to achieve a specific goal.

Examples:

1. Schedule a meeting with my advisor next week
2. Refactor this repository from Java to Python

Long-Horizon Tasks

Long-horizon tasks require:

1. Decomposition
2. State tracking
3. Tool use
4. Branching
5. Error recovery

Long-Horizon Tasks

Schedule a meeting with my advisor next week

- Access my calendar, check when I am free, access my advisor’s calendar, check when my advisor is free, find the intersection of this set, use a calendar tool, send the meeting invite

Long-Horizon Tasks

Refactor this repository from Java to Python

- Understand the repo goal, understand the repo structure, divide repo into standalone refactorable modules, write test cases to ensure correct refactoring, iteratively {refactor module → run tests → edit until tests pass}

METR

https://metr.org/time-horizons/

Measuring AI Ability to Complete Long Software Tasks

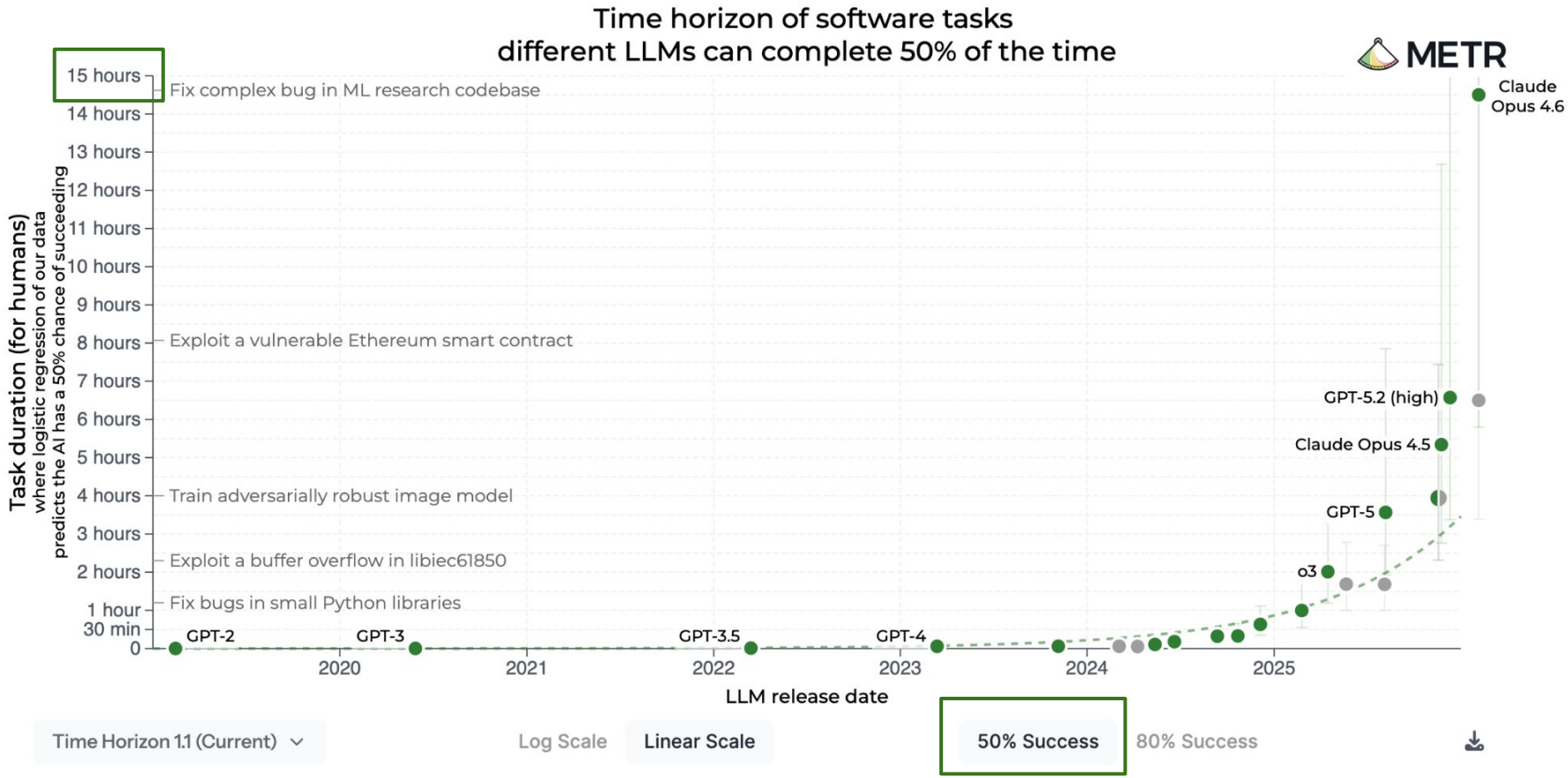
Thomas Kwa*†, Ben West*, Joel Becker, Amy Deng, Katharyn Garcia,
Max Hasin, Sami Jawhar, Megan Kinniment, Nate Rush, Sydney Von Arx

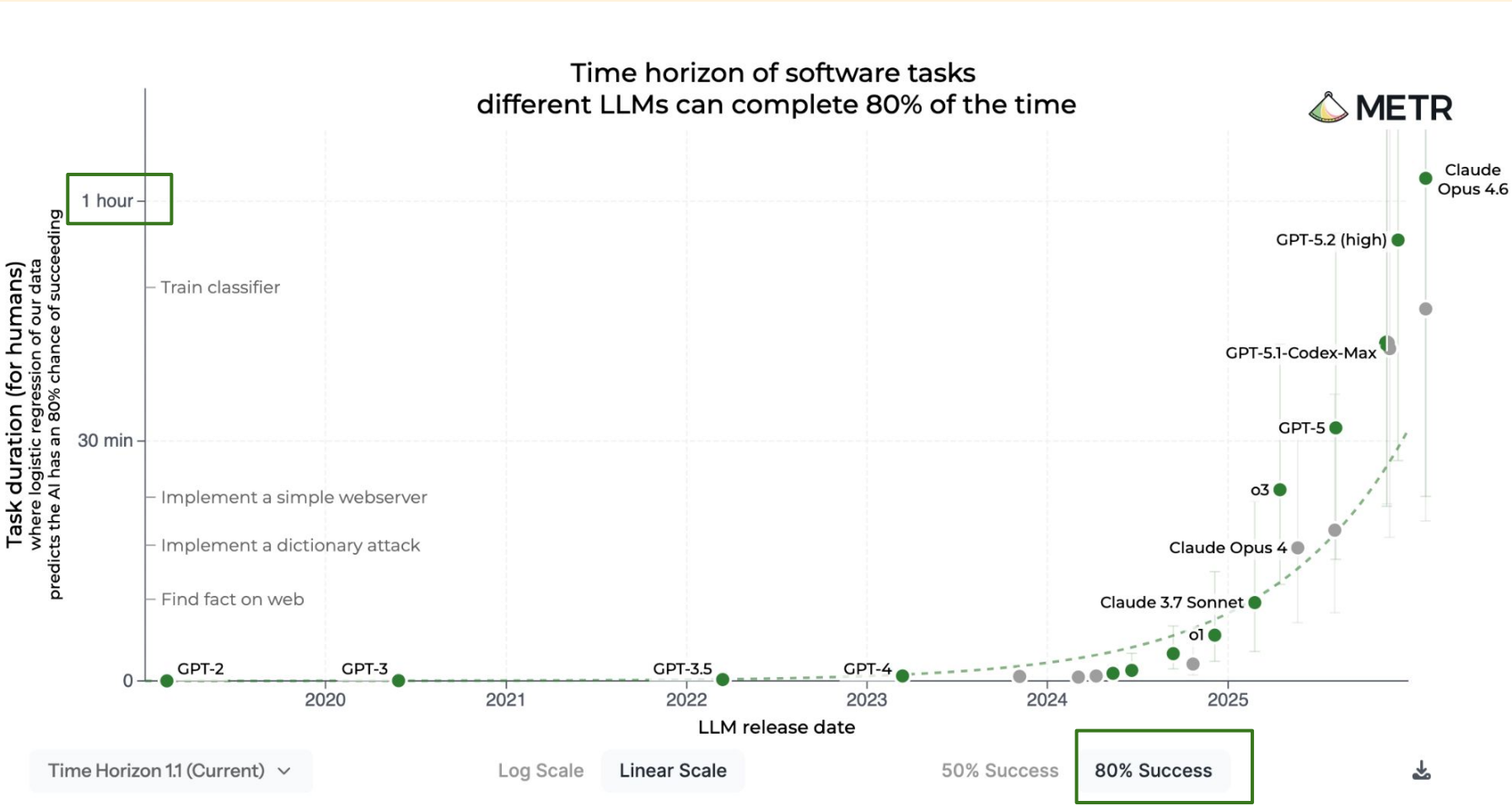
Ryan Bloom, Thomas Broadley, Haoxing Du, Brian Goodrich, Nikola Jurkovic,
Luke Harold Miles‡, Seraphina Nix, Tao Lin, Chris Painter, Neev Parikh, David Rein,
Lucas Jun Koba Sato, Hjalmar Wijk, Daniel M. Ziegler§

Elizabeth Barnes, Lawrence Chan

Model Evaluation & Threat Research (METR)

“To estimate the time horizons of frontier AI agents, we first estimate the duration it takes a human expert to complete each of our tasks. For each agent, we fit a logistic curve to predict the probability it successfully completes tasks as a function of human task duration.”


Time horizon of software tasks
different LLMs can complete 50% of the time
METR
Task duration (for humans)
where logistic regression of our data predicts the AI has a 50% chance of succeeding
15 hours
14 hours
13 hours
12 hours
11 hours
10 hours
9 hours
8 hours
7 hours
6 hours
5 hours
4 hours
3 hours
2 hours
1 hour
30 min
0
Fix complex bug in ML research codebase
Exploit a vulnerable Ethereum smart contract
Train adversarially robust image model
Exploit a buffer overflow in libiec61850
Fix bugs in small Python libraries
GPT-2
GPT-3
GPT-3.5
GPT-4
o3
GPT-5
Claude Opus 4.5
GPT-5.2 (high)
Claude Opus 4.6
2020
2021
2022
2023
2024
2025
LLM release date
Time Horizon 1.1 (Current)
Log Scale
Linear Scale
50% Success
80% Success


Time horizon of software tasks
different LLMs can complete 80% of the time
METR
Task duration (for humans)
where logistic regression of our data
predicts the AI has an 80% chance of succeeding
1 hour
Train classifier
30 min
Implement a simple webserver
Implement a dictionary attack
Find fact on web
0
GPT-2
GPT-3
GPT-3.5
GPT-4
Claude Opus 4.6
GPT-5.2 (high)
GPT-5.1-Codex-Max
GPT-5
o3
Claude Opus 4
Claude 3.7 Sonnet
o1
2020
2021
2022
2023
2024
2025
LLM release date
Time Horizon 1.1 (Current)
Log Scale
Linear Scale
50% Success
80% Success

Architectural Evolution of Agents

Level 0: Single Prompt

Level 1: Prompt Chaining

Level 2: Routing

Level 3: Planning

Each level gives your agent a bit more “control” over the correctness of its output

Prompt Chaining

Prompt chaining decomposes a task into sequential stages:

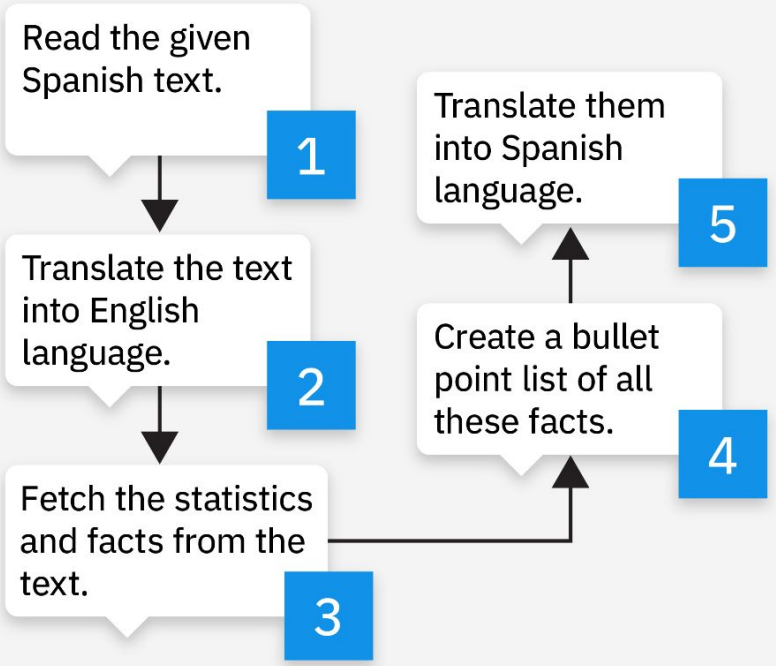
$$Output_i \rightarrow Input_{i+1}$$

It creates a pipeline of transformations, where the output of prompt i serves as the input to prompt i+1

Complex Prompt

Consider the given text in Spanish. Translate it into English. Find all the statistics and facts used in this text and list them as bullet points. Translate them again into Spanish.

Simple Prompt



Example: Document Workflow

Instead of: “Summarize and analyze this report.”

1. Summarize
2. Extract trends
3. Extract evidence
4. Generate executive memo

```
summary = llm("Summarize this document: " + doc)
trends = llm("Extract key trends from: " + summary)
memo = llm("Write executive memo using: " + trends)
```

Chaining Improves Reliability

Smaller prompts:

- Reduce hallucination
- Reduce instruction overload
- Are easier to debug (each stage is inspectable)

Overall, prompt chaining shifts the responsibility from the **user** (to write highly detailed prompts) to the **system** (to take a vague prompt and ensure the output is still correct)

“Transactional Thinking”

Each chaining step is like a local transaction (in distributed systems)

When implementing prompt chaining, keeping track of intermediate states:

```
{
  "trends": [...],
  "risks": [...],
  "recommendations": [...]
}
```

→ easy revising or retrying (upon errors) + easy revision to an earlier state

Limitations of Chaining

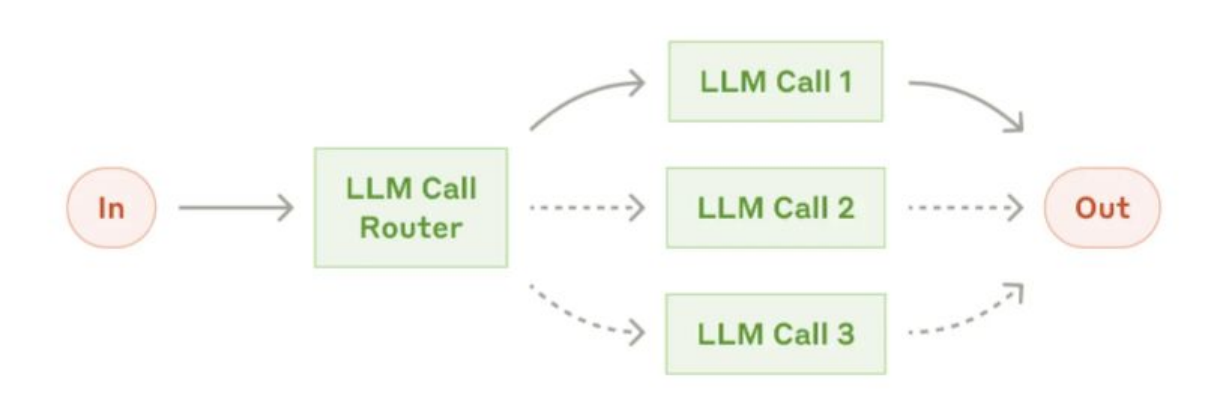
Chaining assumes:

- Known task structure
- Linear progression
- Identical workflow for every input

Real-world tasks are heterogeneous! Different inputs may need different chains

→ **Routing**

Routing



Routing decides:

Which prompt, model, or workflow should handle this input?

It introduces **branching control flow.**

Why Routing is Necessary

Example: Customer support agent

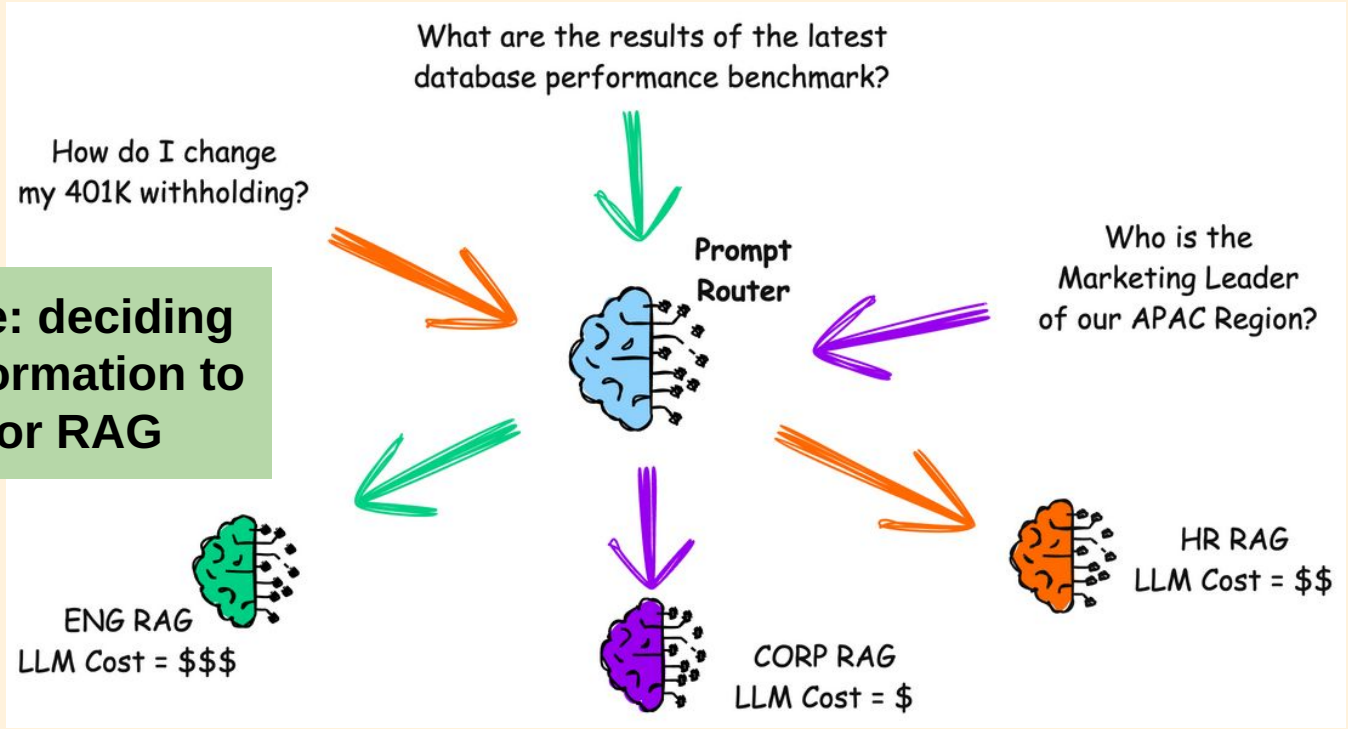
User input could be related to:

- Billing
- Technical bug
- Refund
- Feature inquiry

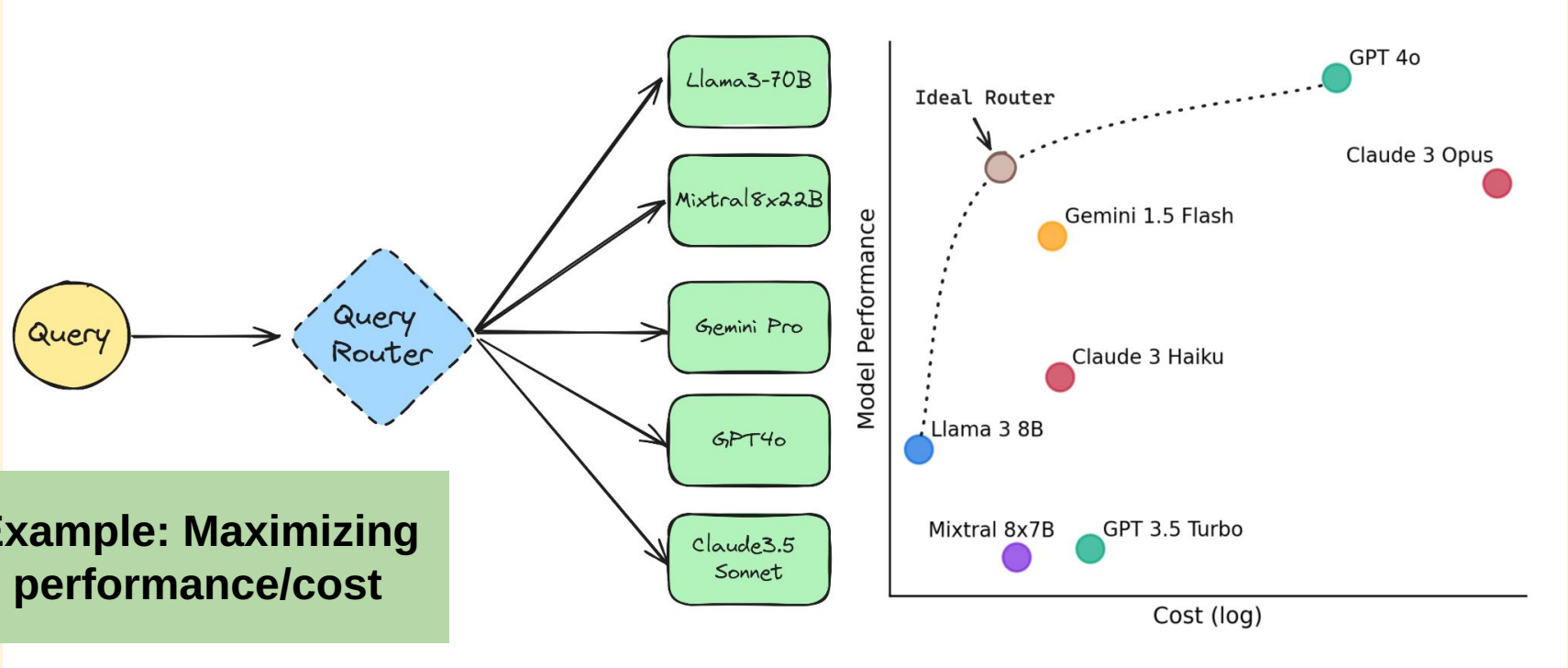
Different problems require different specialized prompts, tools, etc.

Routing can be for prompts, models, or workflows

Example: deciding what information to use for RAG



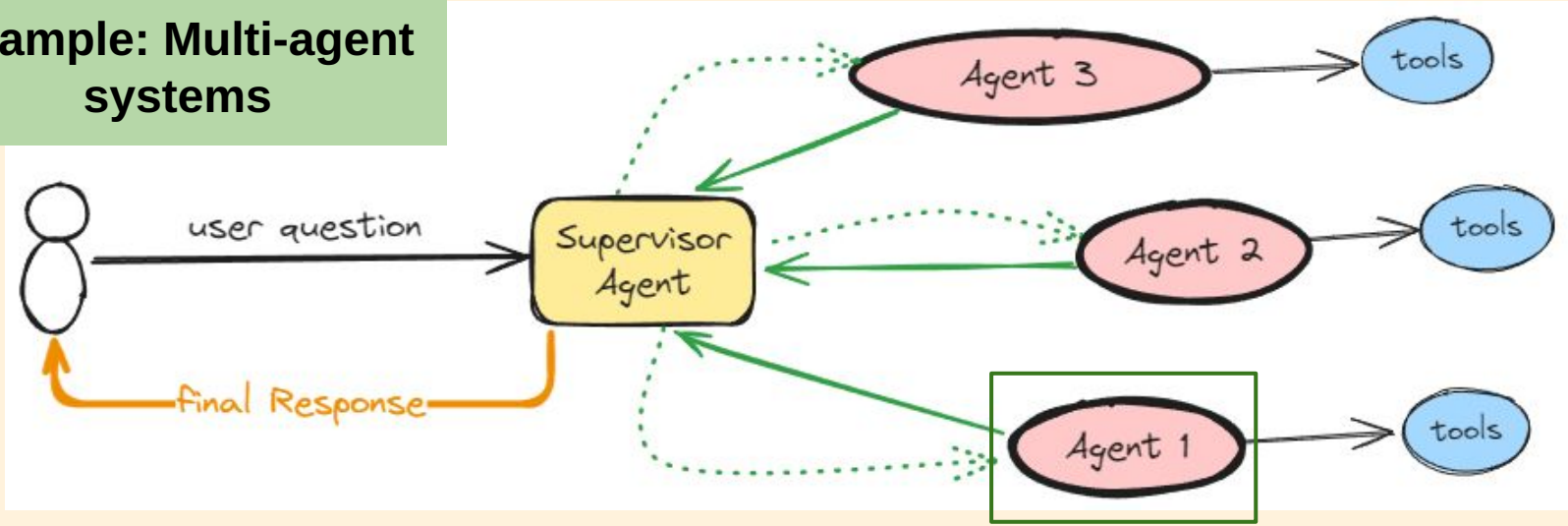
Routing can be for prompts, models, or workflows



Example: Maximizing performance/cost

Routing can be for prompts, models, or workflows

Example: Multi-agent systems



Each agent has its own internal chaining, routing, tool-calling, etc.

Rule-Based Routing

```
if "refund" in query:
    handler = refund_prompt
elif "error" in query:
    handler = technical_prompt
```

LLM-Based Routing

```
category = llm(
  "Classify this query into: [refund, tech, refund, other]: "
   + query
)
```

Embedding-Based Routing

1. Compute embedding of input
2. Compare to category embeddings
3. Select nearest match

Multi-LLM Routing

```
# Route based on: Complexity, Cost, Latency, Specialization
if complexity_score(query) < 0.3:
    model = small_model
else:
    model = large_model
```

Modular Prompt Architecture

Instead of one giant prompt:

→ Create different prompt modules that focus on different specialized tasks

Router dispatches to best module.

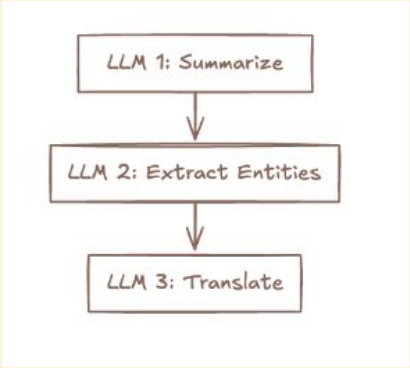
Benefits:

- Maintainability
- Lower token cost
- Independent testing

Recap: Routing vs Chaining

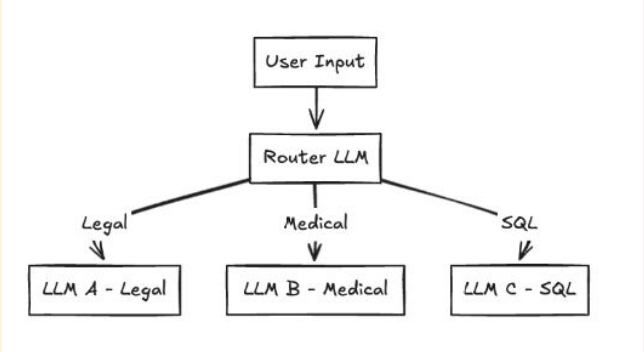
Chaining:

- Linear
- Predictable



Routing:

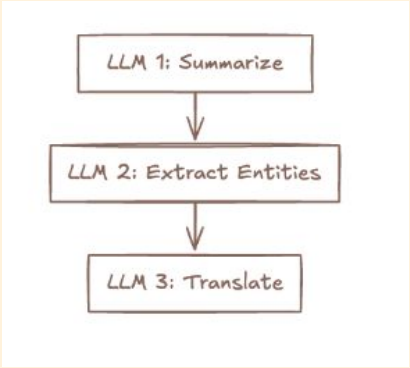
- Branching
- Input-dependent



Recap: Routing vs Chaining

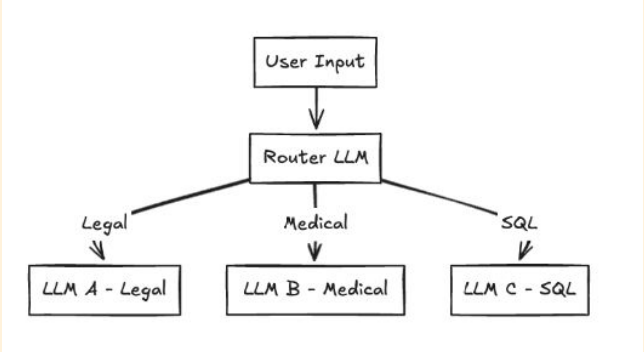
Chaining:

- Linear
- Predictable



Routing:

- Branching
- Input-dependent



Con: Both assume predefined workflows.

Planning

Planning enables goal-directed behavior:

1. Interpret + set goal
2. Decompose into task(s)
3. Execute task(s)
4. Observe results
5. Replan using observations

It introduces **adaptive control.**

Static vs Iterative Planning

Static:

1. Plan once
2. Execute

Iterative (supports uncertainty):

1. Plan partially
2. Execute
3. Adjust

Agentic Planning Loop

An agent integrates planning, routing, tool use, memory, and looping control

```
while not goal_satisfied:
    plan = llm("What should I do next?")
    action = select_tool(plan)
    result = execute(action)
    update_memory(result)
```

Planning vs. ReAct

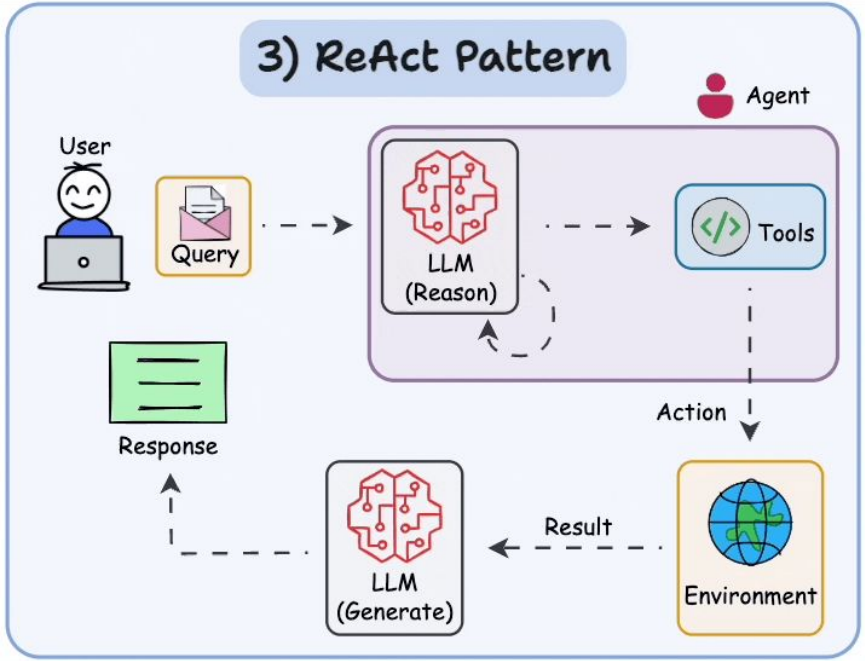
ReAct:

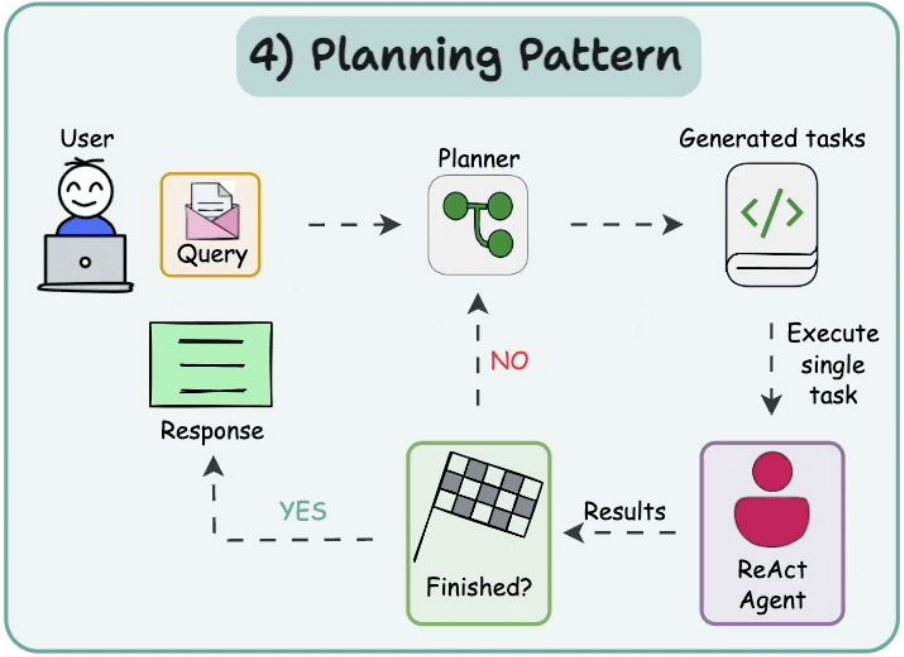
- Reasoning + acting in a tight loop (only plans next action)
- No explicit global plan
  - Used for a single task

Planning:

- Planner generates explicit sequence of tasks before execution
- Used for a large goal containing many tasks (e.g. long-horizon)
  - Tasks are often executed by ReAct Agents

Planning vs. ReAct





Design Principles: Recap

When building an agent…

- Use chaining when structure is predictable + intermediate outputs needed.
- Use routing when inputs vary widely and specialization is important.
- Use static planning when overall goal is complex but easily decomposable.
- Use dynamic planning when overall goal is complex and plans need multiple iterations with intermediate feedback.

Most agents use multiple of these patterns in combination.

Tic-Tac-Toe Agent

Practice chaining & routing!

Download planning.py from the website

Build an agent that follows the diagram on the right