

The Agentic Harness

Shreya Havaladar

Announcements

- **HW 2 is due today**
 - You don't need to ask for permission to use late days!!
- HW 3 will be released tomorrow

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.
- No built-in mechanism for recovering from failure

We need structure around individual model calls.

Single Prompts: Failure Modes

Example task:

“Research competitors to my startup, determine their business models, compare positioning, generate strategy memo for profit.”

How many distinct things are we actually asking the model to do here?

Single Prompts: Failure Modes

Example task:

“Research competitors to my startup, determine their business models, compare positioning, generate strategy memo for profit.”

How many distinct things are we actually asking the model to do here?

Research → ***extract*** → ***compare*** → ***reason*** → ***write***.

Single Prompts: Failure Modes

Example task:

“Research competitors to my startup, determine their business models, compare positioning, generate strategy memo for profit.”

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 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

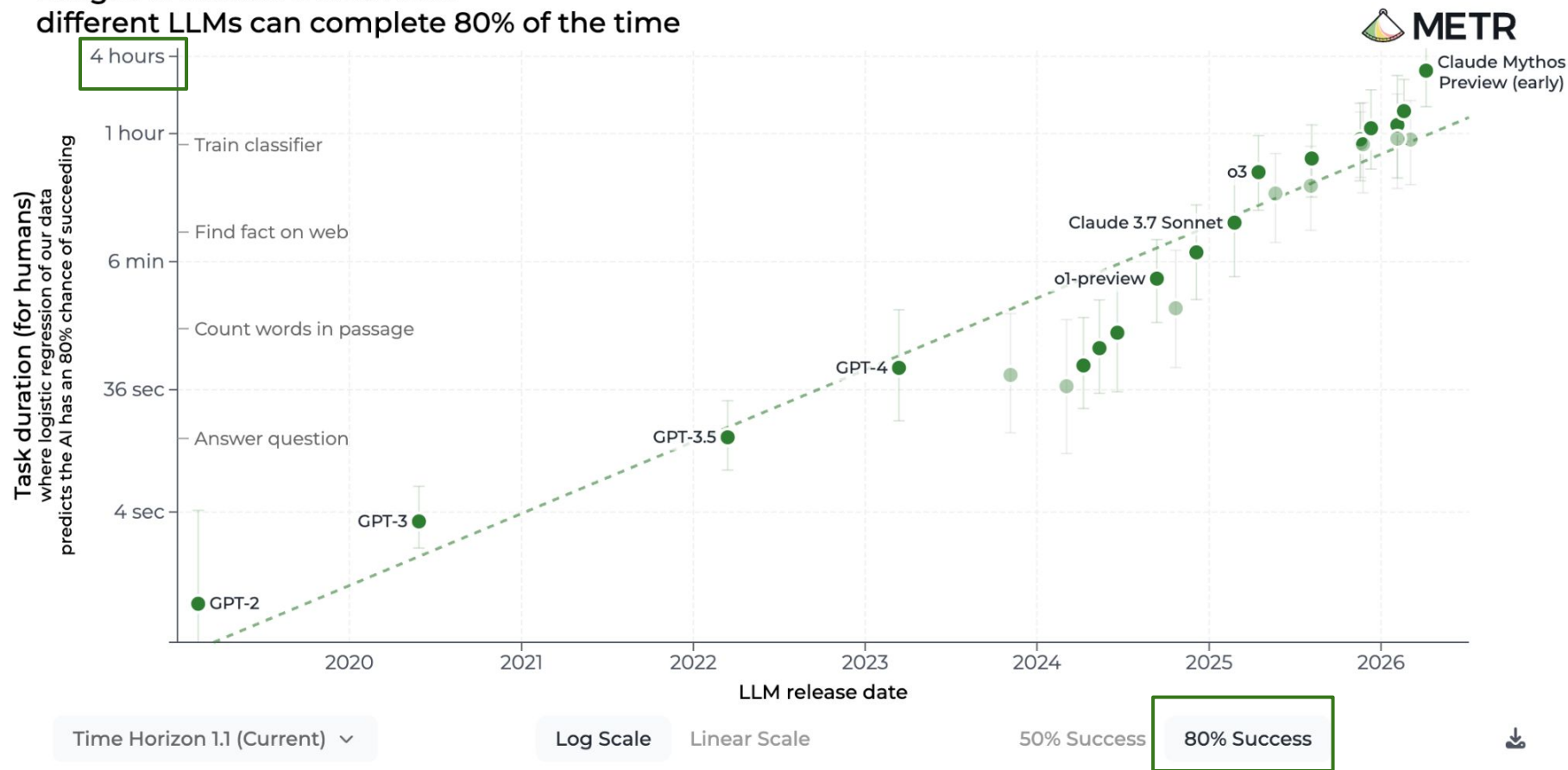
Model Evaluation & Threat Research (METR)

“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.”

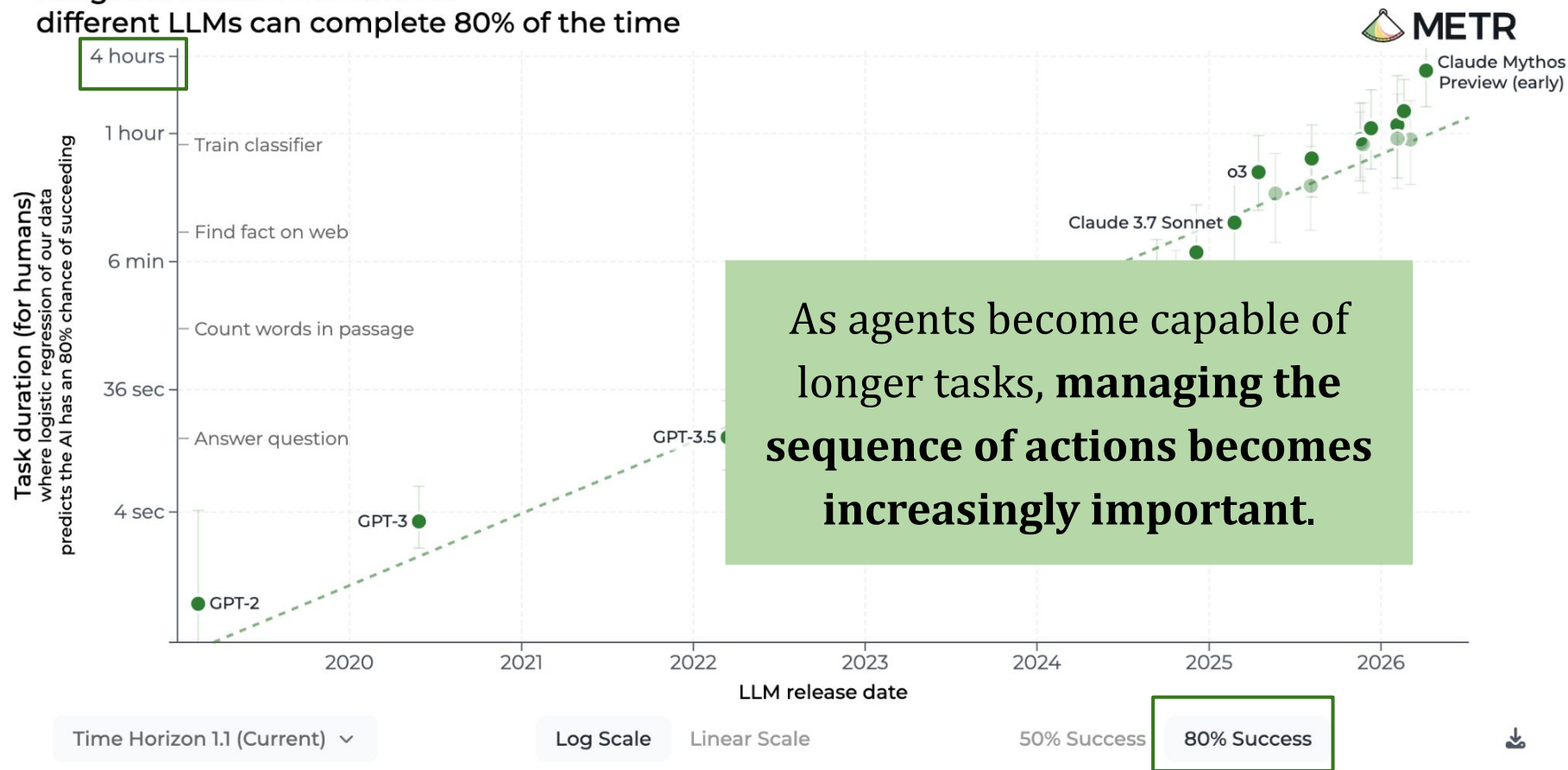
Length of software tasks that different LLMs can complete 50% of the time



Length of software tasks that different LLMs can complete 80% of the time



Length of software tasks that different LLMs can complete 80% of the time



The Agent Harness

An LLM is not an agent

- *it is a function that maps input text $\rightarrow$ output text*

An individual LLM call does not inherently:

- retain persistent task state
- call itself again
- manage a multi-step workflow

Something outside the LLM has to control these things

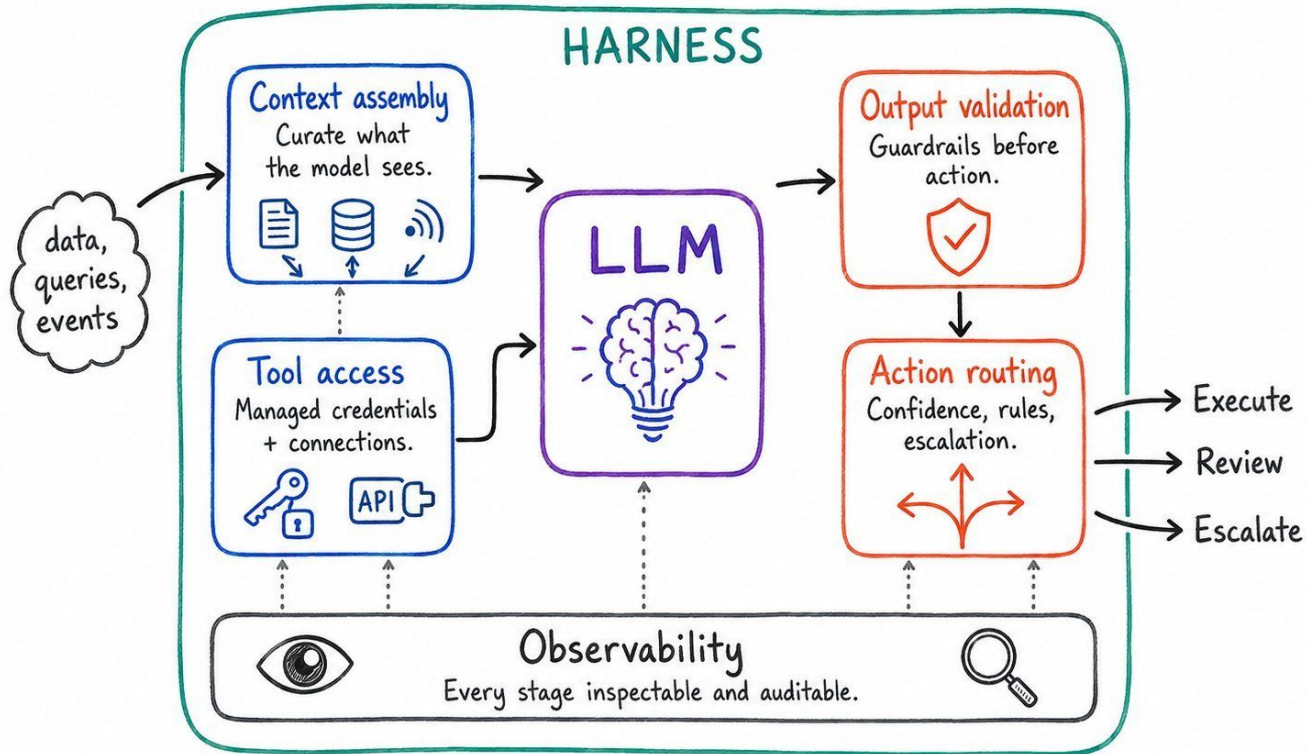
The Agent Harness

Agent harness: the runtime infrastructure surrounding an LLM that turns individual model calls into a stateful system.

The harness is responsible for:

- constructing context
- maintaining task state
- controlling execution
- deciding what happens next
- deciding when to stop

Agent = Model + Harness



The model reasons. The harness governs everything else.

The Harness Loop

1. User request
2. Build context
3. Call LLM
4. Interpret output
5. Update state
6. What happens next?
7. continue / branch / replan / finish

The Harness Loop

1. User request
2. Build context
3. Call LLM
4. Interpret output
5. Update state
6. What happens next?
7. continue / branch / replan / finish

Designing your agent well is
important for this step to run
smoothly!!

State != Context

STATE = Everything the system currently knows

CONTEXT = What this LLM call is allowed to see

You have to determine what parts of state to select and incorporate into the context of the next LLM call

State != Context

STATE = Everything the system currently knows

CONTEXT = What this LLM call is allowed to see

You have to determine what parts of state to select and incorporate into the context of the next LLM call

- You almost never want to include ALL information into the context (token cost, confusion due to extraneous information, etc.)

Static vs. dynamic workflows

Static	Dynamic
Code decided what happens next	Model participates in deciding what happens next
Predefined control flow	runtime control flow
model fills in individual steps	can adapt to observations

Static or Dynamic? (5 min)

A customer support system classifies a user request as billing, looks up the user's account information, and writes a response.

A coding assistant investigates a latency spike, checks metrics and logs, and decides what to inspect next based on what it finds.

You ask a travel planning system for a trip under \$1,000. The system searches options, adjusts the itinerary, and gives you a suggested travel plan.

Static or Dynamic? (5 min)

A customer support system classifies a user request as billing, looks up the user's account information, and writes a response.

Static

A coding assistant investigates a latency spike, checks metrics and logs, and decides what to inspect next based on what it finds.

Dynamic

You ask a travel planning system for a trip under \$1,000. The system searches options, adjusts the itinerary, and gives you a suggested travel plan.

Dynamic

A spectrum of control

More static

More dynamic

Single prompt → Chaining → Routing → Planning → Agentic loop

*Developer specifies
control flow*

*LLM determines
control flow*

A spectrum of control

Single Prompt: no developer involvement

Chaining: developer specifies the path

Routing: developer specifies multiple possible paths

Planning: model creates the path(s)

Agentic loop: model repeatedly plans and acts until goal is reached

Prompt Chaining

Prompt chaining decomposes a task into sequential stages:

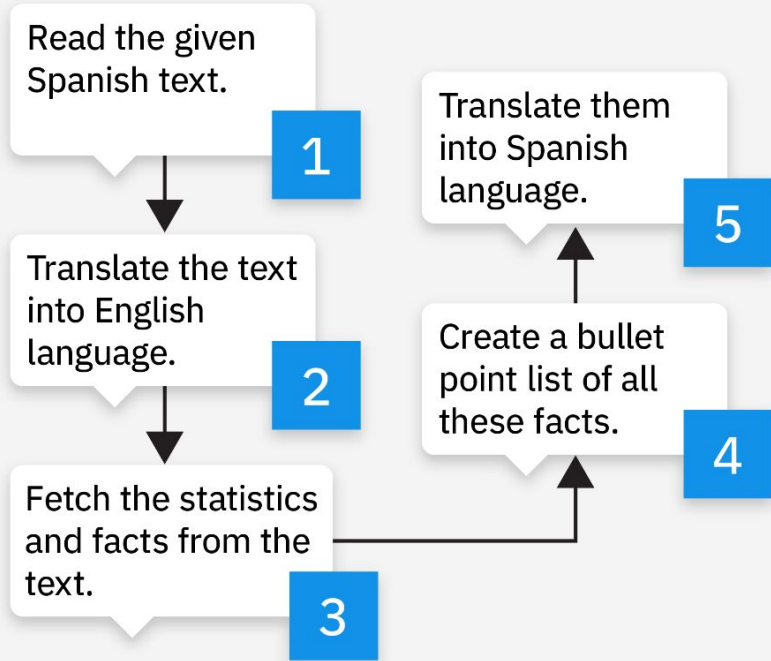
$$Output_i \rightarrow Input_{i+1}$$

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



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

Read the given
Spanish text.

1

Translate them
into Spanish

Trans
int
lan

We're moving complexity
out of a single generation
and into the application's
architecture.

5

4

Fetch the statistics
and facts from the
text.

3

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)
```

Intermediate outputs are useful

When implementing prompt chaining, keeping track of intermediate states:

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

1. Easy revising or retrying (upon errors)
2. Easy revision to an earlier state

Prompt Chaining $\neq$ Chain-of-Thought

Chain-of-thought

ONE model call $\rightarrow$ model reasons $\rightarrow$ answer

Prompt chaining

model call $\rightarrow$ program receives output $\rightarrow$ model call $\rightarrow$
program receives output $\rightarrow$... $\rightarrow$ answer

**Prompt chaining is an application architecture pattern, not a prompting technique.*

Chaining Improves Reliability

Smaller stages:

- reduce instruction overload
- isolate responsibilities
- make intermediate results visible
- make failures easier to localize

Chaining doesn't prevent errors. But, it does make errors easier to **find** and **recover from**.

Limitations of Chaining

Failure Propagates Through a Chain

Decomposition creates inspectable failure points, but downstream steps still inherit upstream mistakes.

Chaining assumes:

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

Limitations of Chaining

Chaining is only applicable to a subset of tasks:

- there is a natural sequence of transformations
- later steps genuinely depend on earlier outputs
- intermediate results are useful artifacts

Probably don't chain:

`“Rewrite this paragraph more concisely”`

Probably do chain:

`“Read 100 reviews → extract complaints → group themes → write report”`

Limitations of Chaining

Chaining is only applicable to a subset of tasks:

- there is a natural sequence of transformations
- later steps genuinely depend on earlier outputs
- intermediate results are useful artifacts

Probably don't chain:

`“Rewrite this paragraph more concisely”`

Probably do chain:

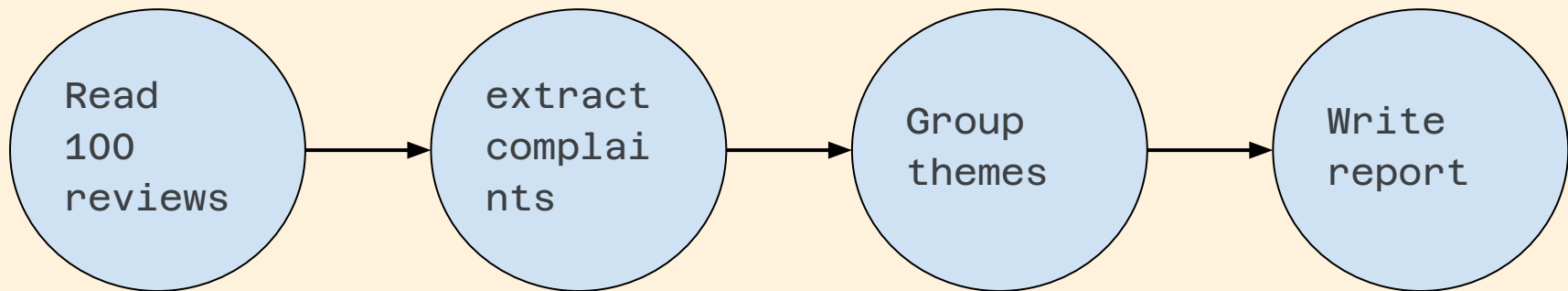
`“Read 100 reviews → extract complaints → group themes → write report”`

Control Flow as a Graph

Node = computation

Edge = what happens next

State = information carried through the graph

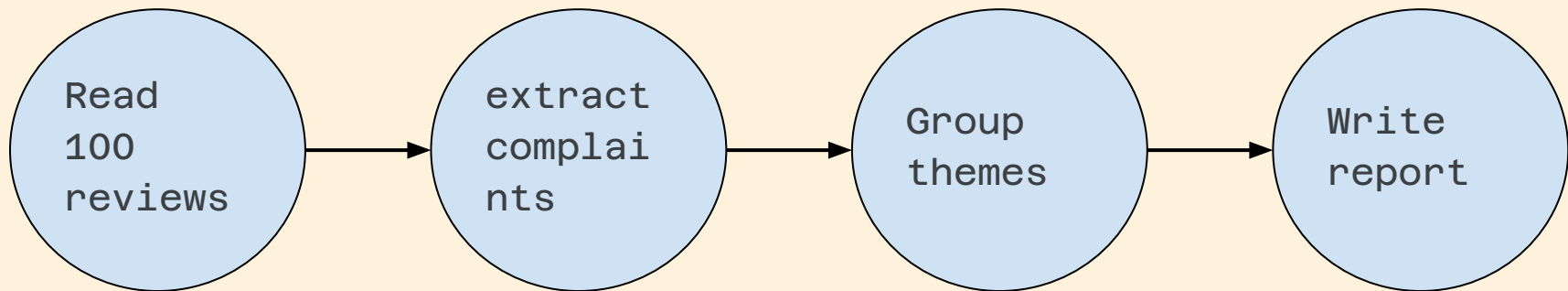


Control Flow as a Graph

Node = computation

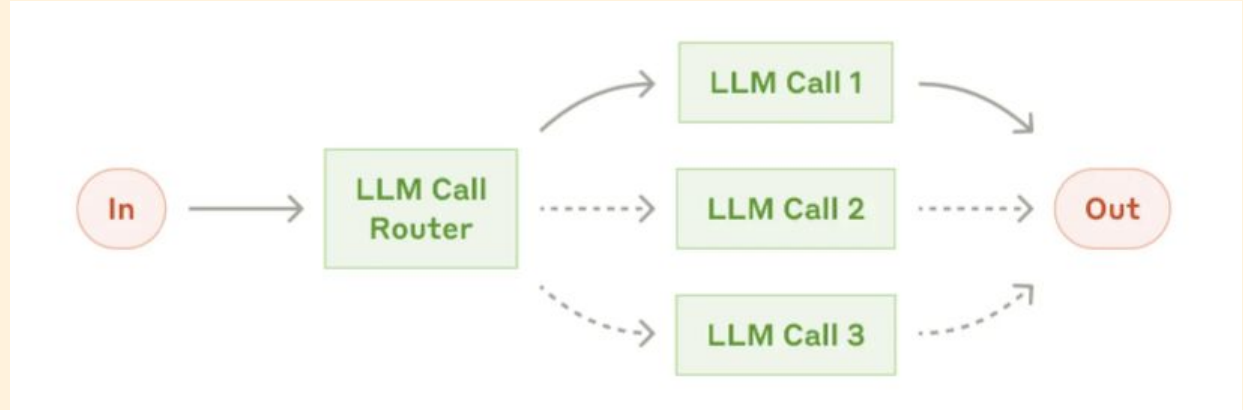
Edge = what happens next

State = information carried through the graph



What happens if a node can have multiple possible next steps?

Routing



Routing decides:

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

It introduces **branching control flow**.

Why Routing is Necessary

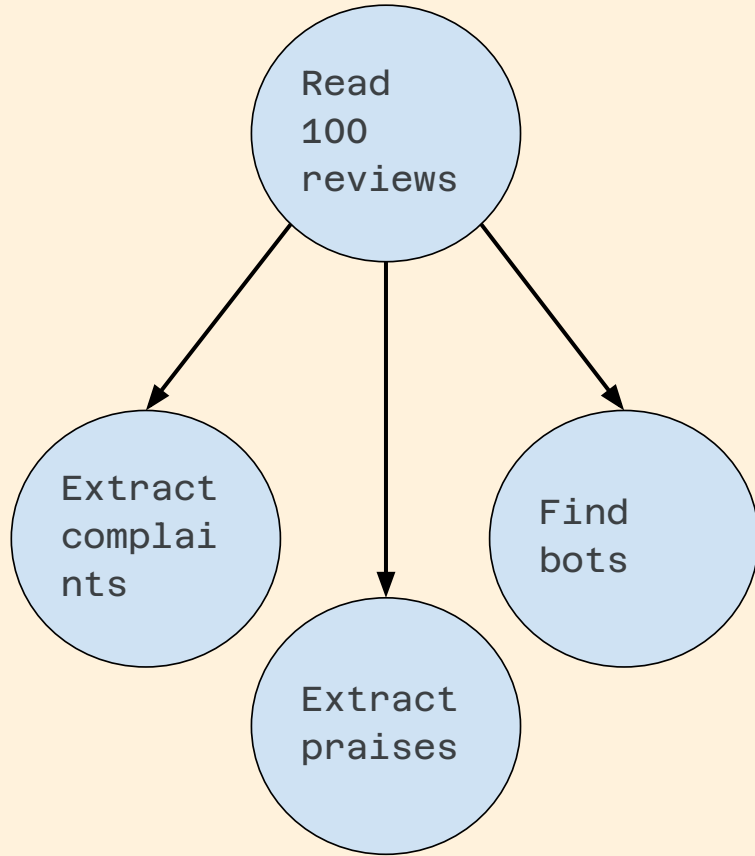
Example: Customer support agent

User input could be related to:

- Billing
- Technical bug
- Refund
- Feature inquiry

Different problems require different specialized prompts, tools, etc.



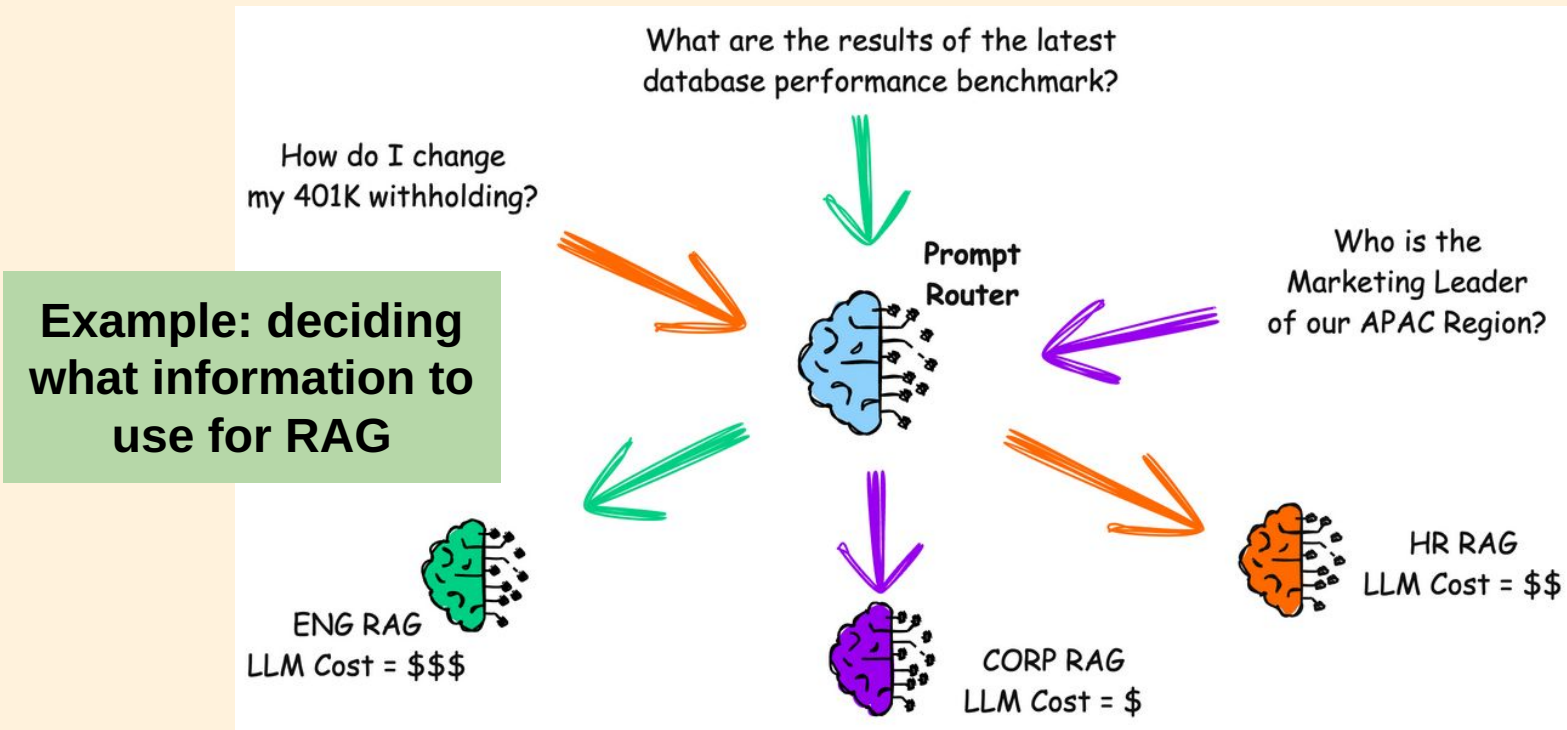


Routing Chooses Among Predefined Paths

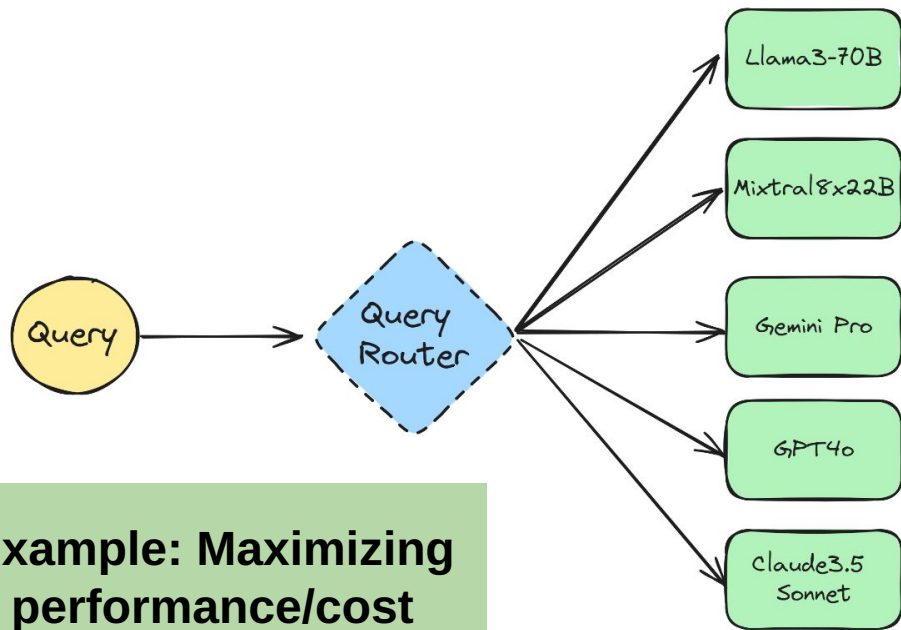
Developer still determines:

1. What routes exist
2. What each route means
3. What each route does

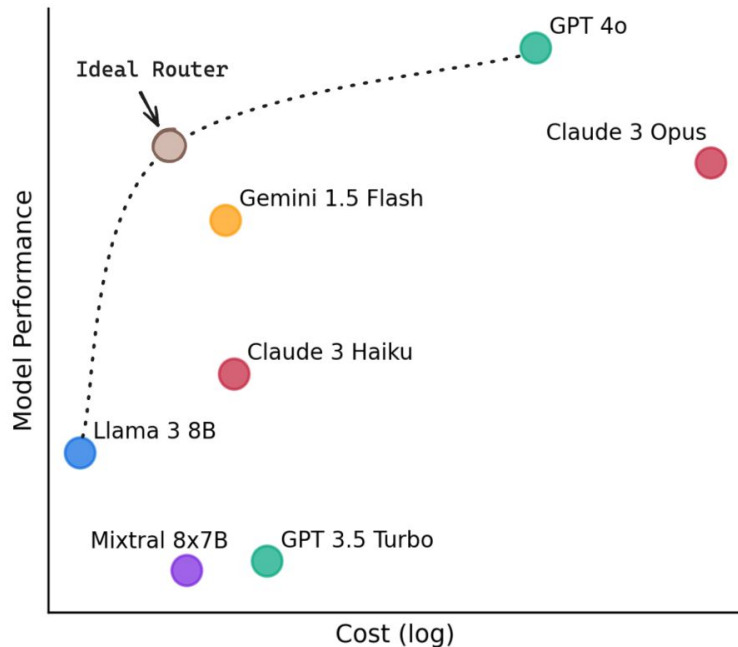
Routing can be for prompts, models, or workflows



Routing can be for prompts, models, or workflows

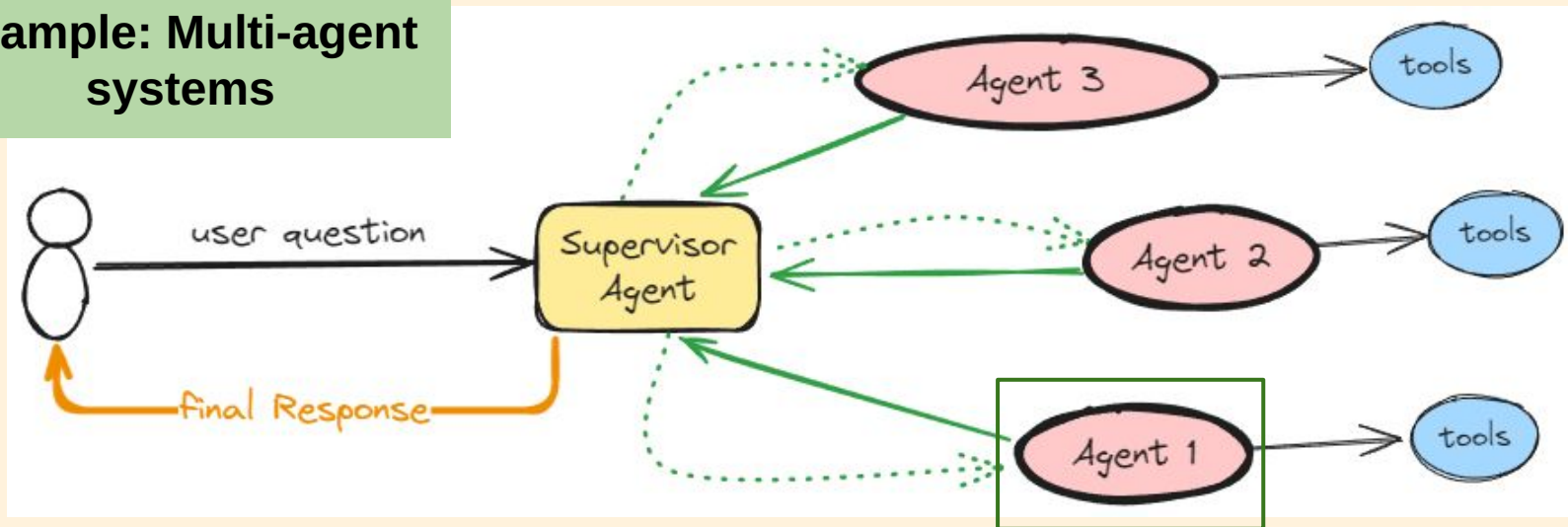


Example: Maximizing performance/cost



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
```

Routing is a classification problem

$r(x) \rightarrow y$

- (x) : current input/state
- (y) : destination

Rule, embedding, and LLM-based routing differ mainly in how we implement (r) .

Routing is a classification problem

Routes: FAQ pages to feed into the LLM based on an input query

1. Billing
2. Refund
3. Account

Input: “I cancelled yesterday but was charged this morning. Can I get my money back?”

What FAQ page should we use?

Routing is a classification problem

Routes: FAQ pages to feed into the LLM based on an input query

1. Billing
2. Refund
3. Account

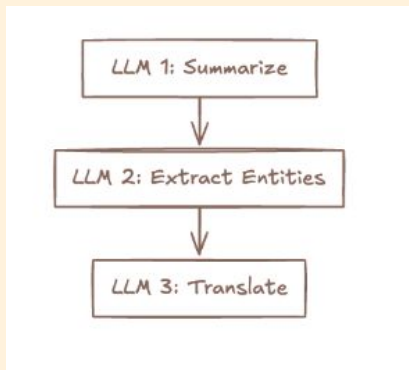
Input: “I cancelled yesterday but was charged this morning. Can I get my money back?”

Routing becomes difficult when categories are not mutually exclusive.

Recap: Routing vs Chaining

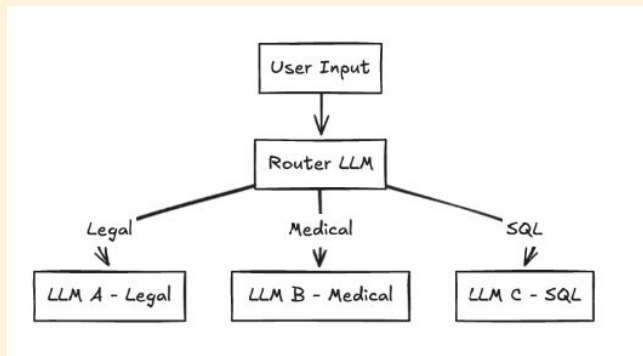
Chaining:

- Linear
- Predictable



Routing:

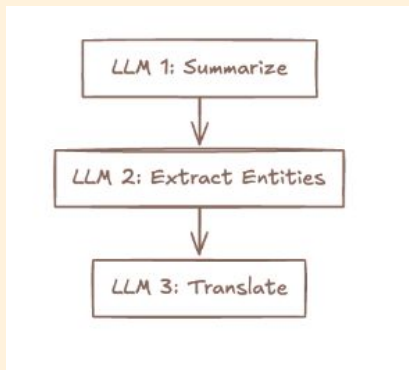
- Branching
- Input-dependent



Recap: Routing vs Chaining

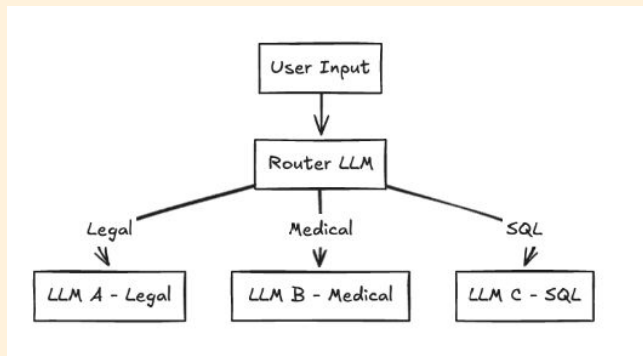
Chaining:

- Linear
- Predictable



Routing:

- Branching
- Input-dependent



**Con: Both
assume
predefined
workflows.**

Planning

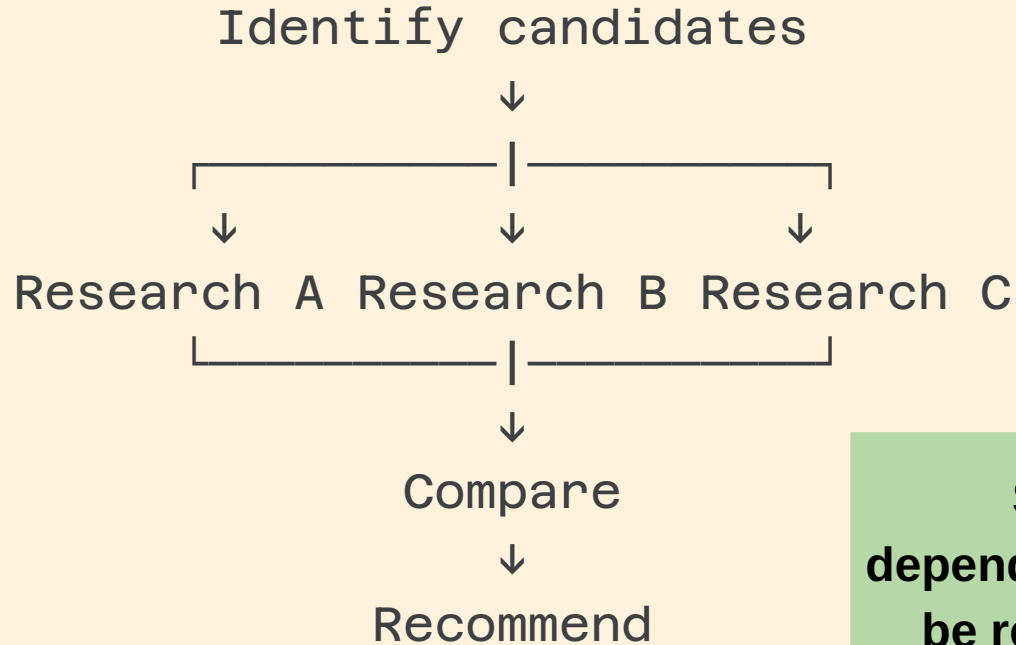
Planning enables goal-directed behavior:

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

It introduces **adaptive control**.



A plan is not always a list



Some tasks have dependencies. So, a plan may be represented as a DAG

Iterative Planning

Standard:

1. Plan once
2. Execute

Iterative (supports uncertainty):

1. Plan partially
2. Execute
3. observe
4. 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) #can involve prompt chaining  
    update_memory(result)
```

Planning vs. ReAct

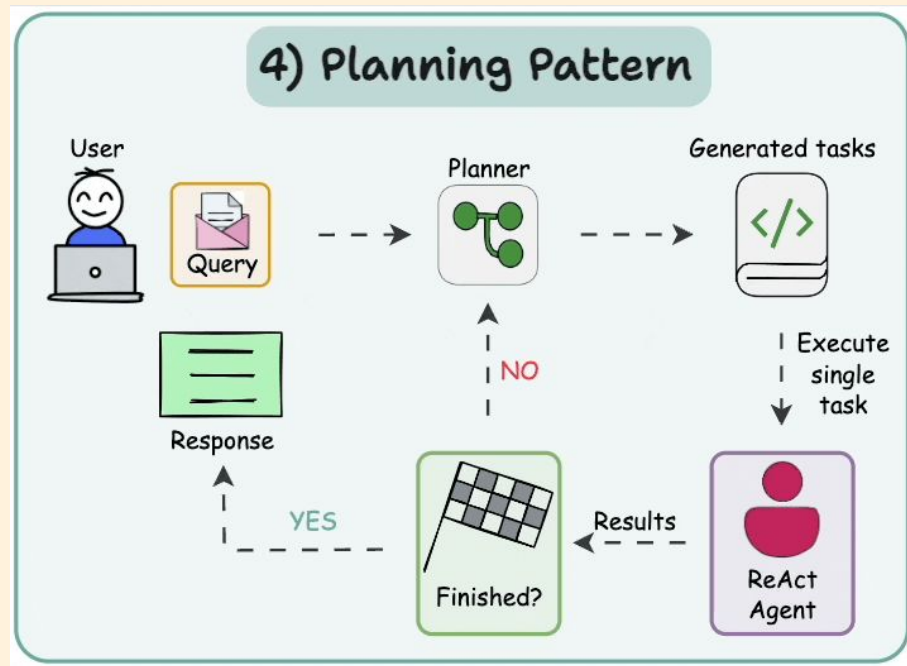
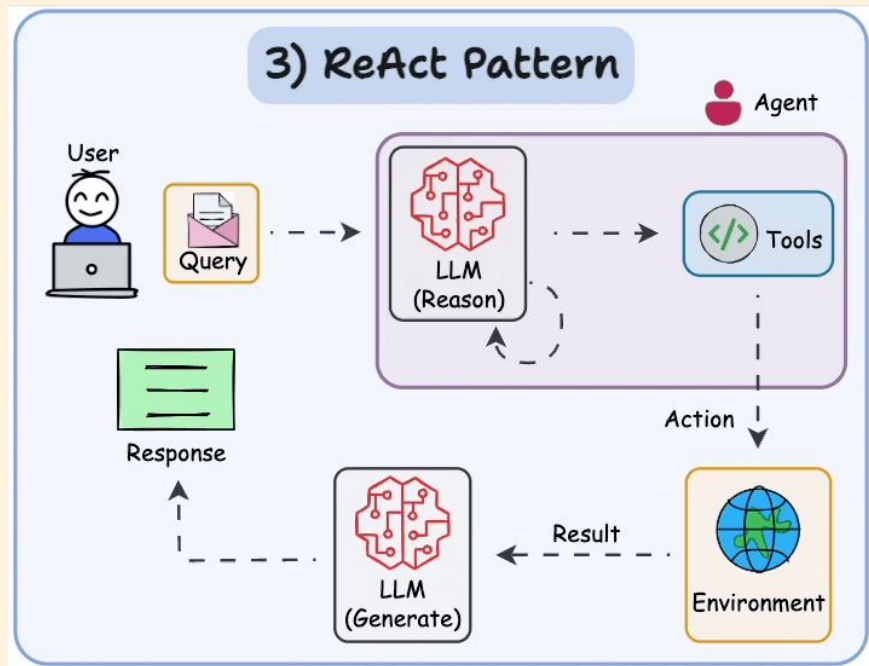
ReAct:

- 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



Planning Strategies

Strategy	Replan when?
Plan-then-execute	Never
Adaptive	Something goes wrong
Continuous	After every observation
Hierarchical	When a sub-plan is needed/completed

Plan-then-execute

Goal

↓

Generate plan

↓

1 → 2 → 3 → 4 → answer

Advantages:

- Simple
- Less repeated deliberation
- Global structure

Weakness:

- The entire plan is based only on information available before execution begins.
- Brittle and cannot adapt

Plan-then-execute

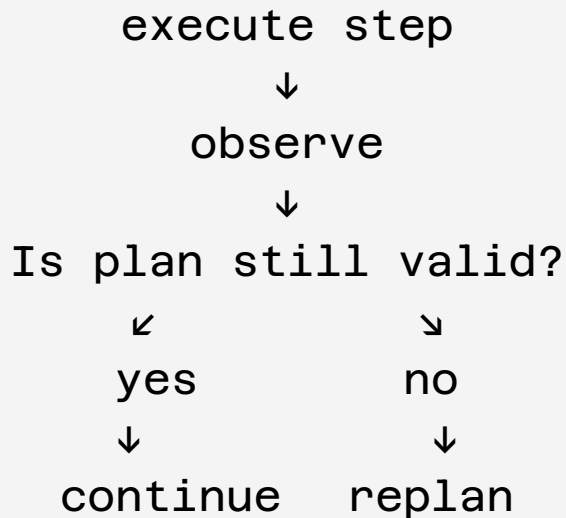
Initial plan: Find the cheapest nonstop flight from NYC to London.

1. Check airline A
2. Check airline B
3. Compare
4. Book cheapest

Observation: Airline A has no nonstop service that day.

Execution reveals information the planner did not have.

Adaptive Planning



Advantages:

- Keeps global structure
- Can adapt to unknown information

Weakness:

- Must detect *when* to replan
- Can continue with a bad plan for too long

Continuous Planning

observe

↓

reconsider

↓

choose next action

↓

observe

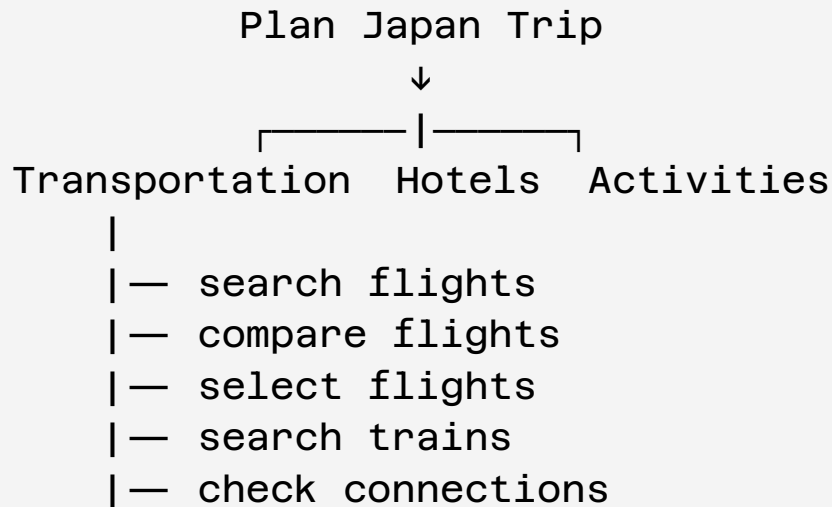
Advantages:

- Maximum adaptability – incorporates every new observation

Weakness:

- more deliberation
- more latency/cost
- easier to “wander”
 - LLM may lose sight of final goal, since it only ever plans the immediate next step

Hierarchical Planning



Advantages:

- Plan high-level first, then plan each subtask only when needed
- Saves time + avoids over-planning

Weakness:

- More complex to implement
- Mistakes in high-level plan can propagate down into subplans

How big should a planning step be?

Goal: Build and launch a course website.

Too coarse	Too broad	Useful
1. Build website 2. Launch	1. Open editor 2. Create file 3. Type <code><h></code> 4. Type <code>
</code> ...	1. Define site structure 2. Implement pages 3. Add content 4. Verify navigation 5. Deploy

**** A useful step should be independently executable and produce an observable result.***

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.