Bhaswata Choudhury
C-DAC

About the speaker

Bhaswata Choudhury

- GenAI Engineer(1.5+ years) at C-DAC Blr
- Experience leading the design and scaling of autonomous AI Agent systems.
- Passionate about the intersection of Software Engineering Best Practices and Large Language Models.

/in/bhaswata-choudhury

bhaswata08

bhaswata08

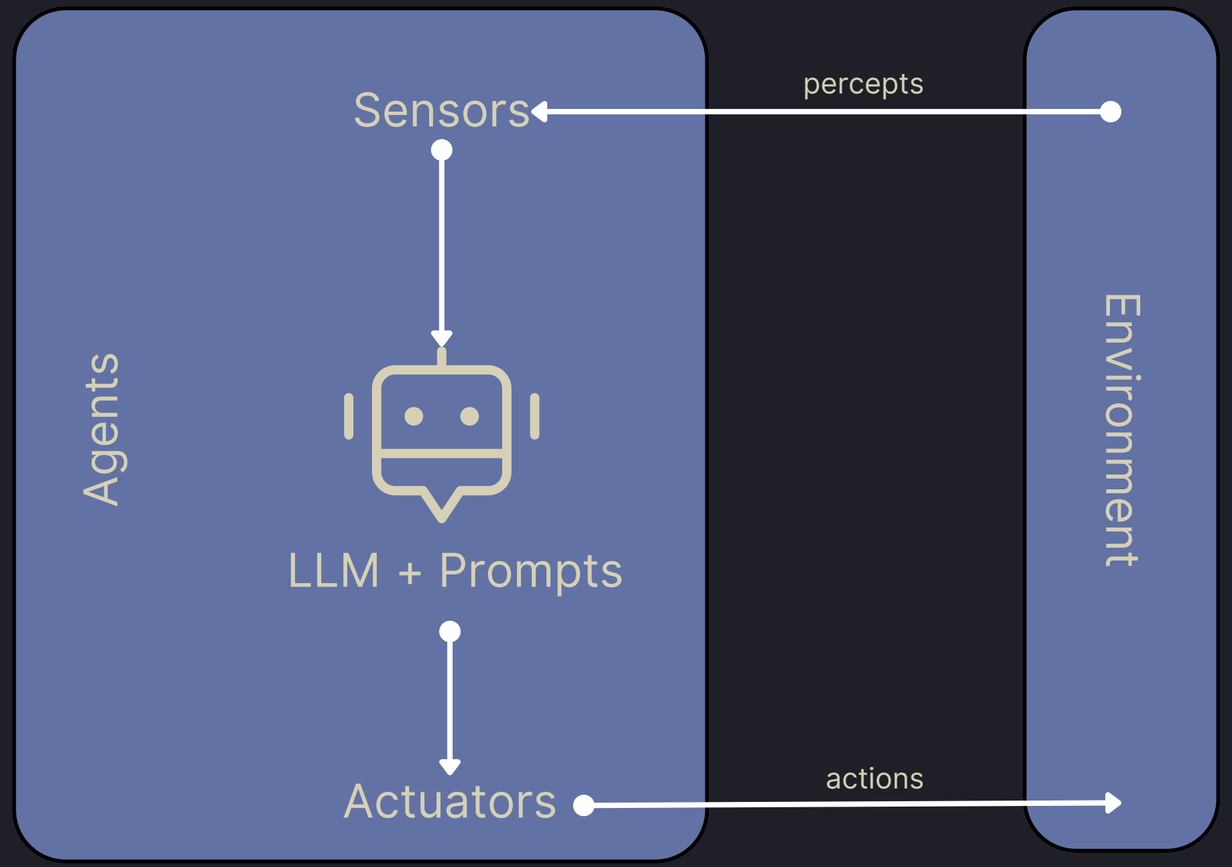
DISCLAIMER

AI WORLD MOVES TOO FAST

LLMs on a Mission

सी डैक
CDAC

What Agents

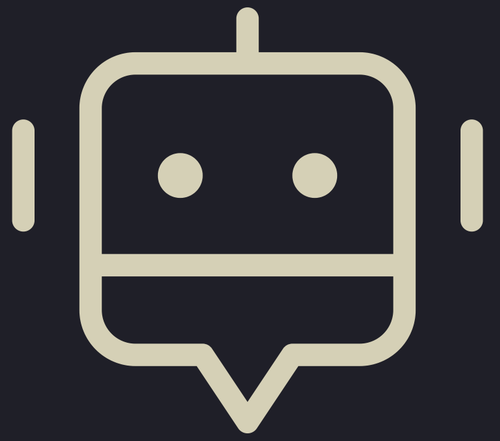

Agents
Sensors
percepts
LLM + Prompts
Actuators
actions
Environment

Tool

Tool

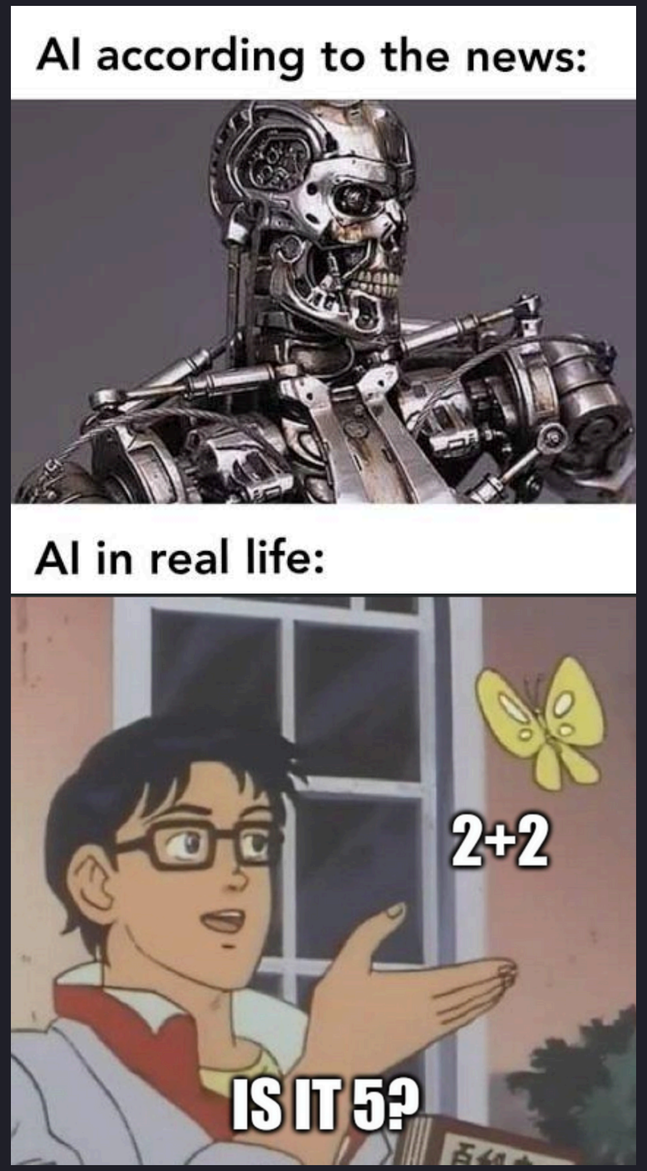
Tool

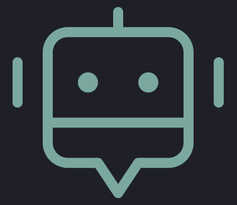
```
@dataclass
class Add:
  a: int
  b: int
```



A Common misconception:

- AI Agents cannot execute tools, they can only output text, tools are executed by automated systems and APIs
- In Tool calling, even simple addition isn’t performed by an LLM
- Remember LLM isnt an Calculator, Its just a text generator that takes in unstructured information and spits out structured information (in tool calling).





LLM

Tools

Architecture

Memory

रीडेक
CDAC

Agentic AI for real world applications

From prototype to production

सी डैक
CDAC

The Agent Journey

1. Decide that you want to build an Agent
2. Product design, what problems to solve
3. Want to move fast so grab $FRAMEWORK and *get to building*
4. Get to 70-80% quality bar
5. Realize that 80% isn’t good enough
6. Realize that getting past 80% requires reverse-engineering the framework, prompts, flow etc.

The Agent Journey

1. Decide that you want to build an Agent
2. Product design, what problems to solve
3. Want to move fast so grab $FRAMEWORK and *get to building*
4. Get to 70-80% quality bar
5. Realize that 80% isn’t good enough
6. Realize that getting past 80% requires reverse-engineering the framework, prompts, flow etc.
7. Start from scratch
8. Realize that this isn’t a good problem for agents!

Not every problem needs an agent

सी डैक
CDAC

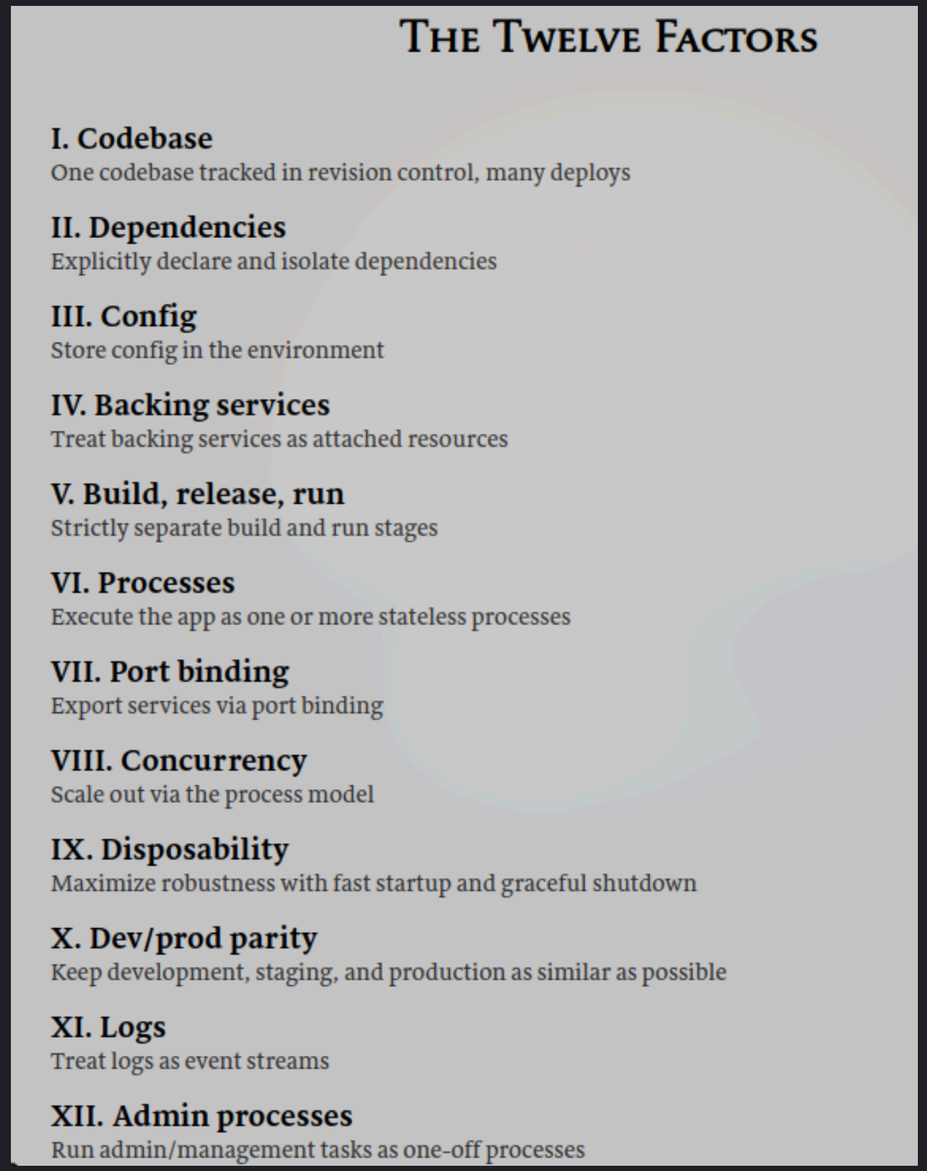
Principles of Reliable LLM Applications

1. There are certain core things that make agents great.
2. Apply small, modular concepts to existing code.
3. This is just software engineering 101

सी डैक
CDAC

12 Factor Agents

Principles of Reliable LLM Applications



The Twelve Factors

I. Codebase
One codebase tracked in revision control, many deploys

II. Dependencies
Explicitly declare and isolate dependencies

III. Config
Store config in the environment

IV. Backing services
Treat backing services as attached resources

V. Build, release, run
Strictly separate build and run stages

VI. Processes
Execute the app as one or more stateless processes

VII. Port binding
Export services via port binding

VIII. Concurrency
Scale out via the process model

IX. Disposability
Maximize robustness with fast startup and graceful shutdown

X. Dev/prod parity
Keep development, staging, and production as similar as possible

XI. Logs
Treat logs as event streams

XII. Admin processes
Run admin/management tasks as one-off processes

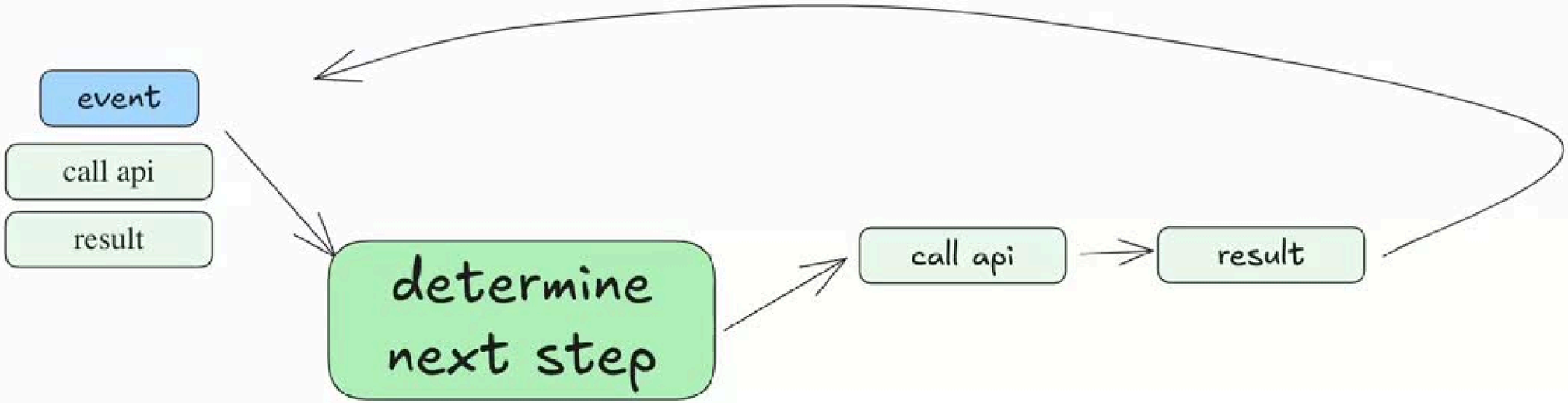
I want you to

Open up the black box

Be creative
Art == Science

Build great agents with sound engineering principles

सी डैक
CDAC


event
call api
result
determine next step
call api
result

LLM Optimization

- **Factor 1:** tool calling
- **Factor 2:** Documentation of tools
- **Factor 3:** Prompt Engineering
- **Factor 4:** Context engineering

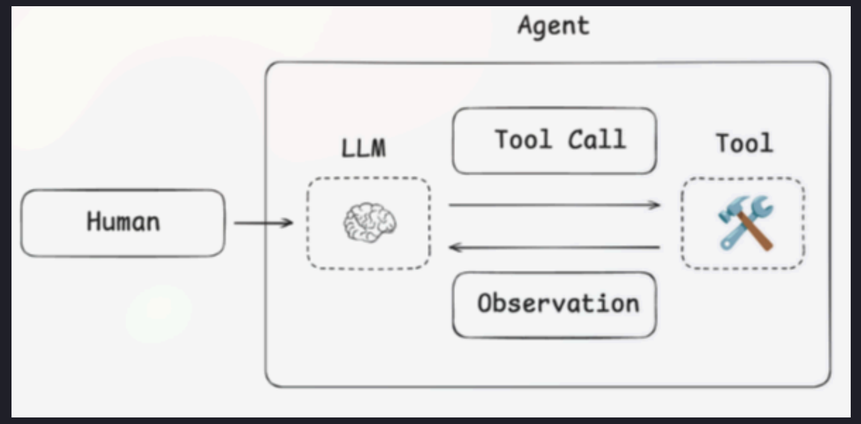
- **Factor 5:** One single logbook

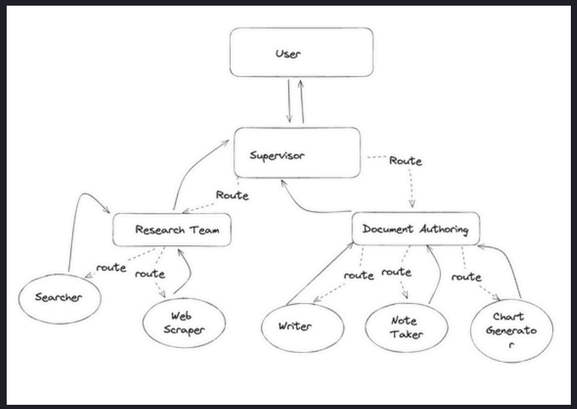
Architecture optimization

- **Factor 6:** Play pause resume time travel
- **Factor 7:** Keep human in Loop
- **Factor 8:** Own your control flow
- **Factor 9:** Compact errors into context window
- **Factor 10:** Build small focused agents

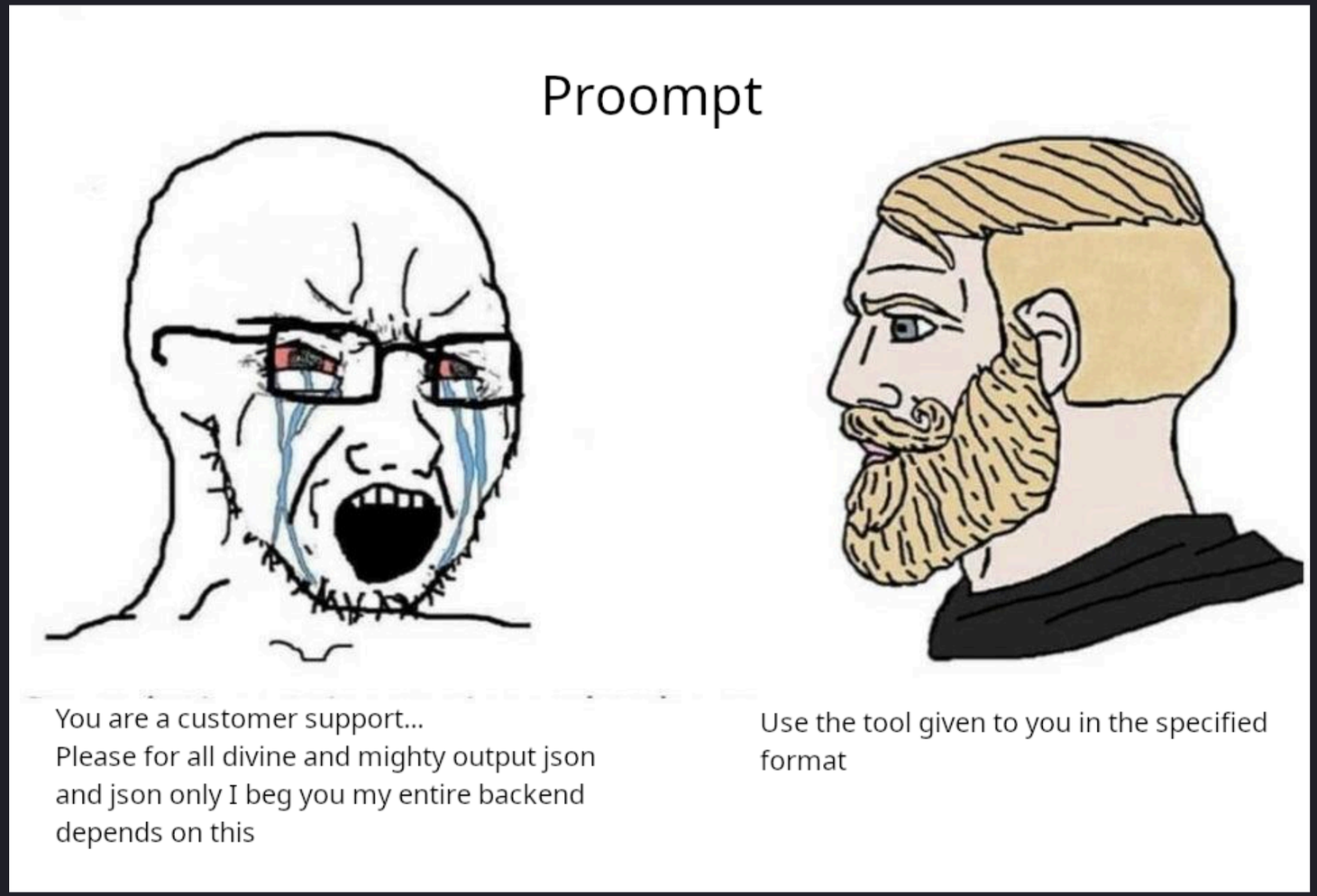
UX Interaction

- **Factor 11:** Meet users where they are
- **Factor 12:** Stateless reducers





FACTOR 1: Use tool calls



Goes to model context as schema to output

```python
from langchain.tools import tool

@tool
def search_database(sql_query: str, limit: int = 10)->str:
    """
    Search the customer SQL database for records matching the query.

    Args:
        query: Search terms to look for
        limit: Maximum number of results to return
    """
    results = call_api(query)
    return f"Found {limit} results for '{query}'"
```

Goes to model context as tool description

सी डैक
CDAC

```python
from pydantic import BaseModel, Field
from typing import Literal

class WeatherInput(BaseModel):
    """Input for weather queries."""
    location: str = Field(description="City name or coordinates")
    units: Literal["celsius", "fahrenheit"] = Field(
        default="celsius",
        description="Temperature unit preference"
    )
    include_forecast: bool = Field(
        default=False,
        description="Include 5-day forecast"
    )

@tool(args_schema=WeatherInput)
def get_weather(location: str, units: str = "celsius", include_forecast: bool = False) -> str:
    """Get current weather and optional forecast."""
    temp = 22 if units == "celsius" else 72
    result = f"Current weather in {location}: {temp} degrees {units[0].upper()}"
    if include_forecast:
        result += "\nNext 5 days: Sunny"
    return result
```

You can use advanced schema objects as well.

AT THE END OF THE DAY:

Tool calls are just structured JSON

1. LLM outputs structured JSON
2. Deterministic code executes the appropriate action (like calling an external API)
3. Results are captured and fed back into the context

```ts
const openai_tools: ChatCompletionTool[] = [
  {
    type: "function",
    function: {
      name: "multiply",
      description: "multiply two numbers",
      parameters: {
        type: "object",
        properties: {
          a: { type: "number" },
          b: { type: "number" },
        },
        required: ["a", "b"],
      },
    },
  },
  {
    type: "function",
    function: {
      name: "add",
      description: "add two numbers",
      parameters: {
        type: "object",
        properties: {
          a: { type: "number" },
          b: { type: "number" },
        },
        required: ["a", "b"],
      },
    },
  },
];
```

JSON Schema

Prompt-Only

```
What should the next step be?

Answer in JSON using any of these schemas:
{
  // you can request more information from me
  intent: "request_more_information",
  message: string,
} or {
  intent: "create_issue",
  issue: {
    title: string,
    description: string,
    team_id: string,
    // name of the team to create the issue in
    team_name: string or null,
    project_id: string or null,
    // name of the project to create the issue in
    project_name: string or null,
    assignee_id: string or null,
    // name of the user to assign the issue to
    assignee_name: string or null,
    labels_ids: string[],
    labels_names: string[],
    // The priority of the issue.
    // 0 = No priority, 1 = Urgent, 2 = High, 3 = Normal, 4 = Low.
    priority: int or null,
  },
} or {
  intent: "list_teams",
  // what is your goal or desired outcome with this operation
  query_intent: string or null,
} or {
```

रीडेक CDCC

Prompting vs JSON Mode vs Function Calling vs Constrained Generation vs SAP

A technical explanation of every way to extract structed data from an LLM

BAML Blog / Jul 29, 2024

noamgat/lm-format-enforcer: Enforce the output format (JSON Schema, Regex et...

Enforce the output format (JSON Schema, Regex etc) of a language model - noamgat/lm-format-enforcer

GitHub

hinthornw/**trustcall**

Tenacious tool calling built on LangGraph

7 Contributors | 712 Used by | 959 Stars | 75 Forks

hinthornw/trustcall: Tenacious tool calling built on LangGraph

Tenacious tool calling built on LangGraph. Contribute to hinthornw/trustcall development by creating an account on GitHub.

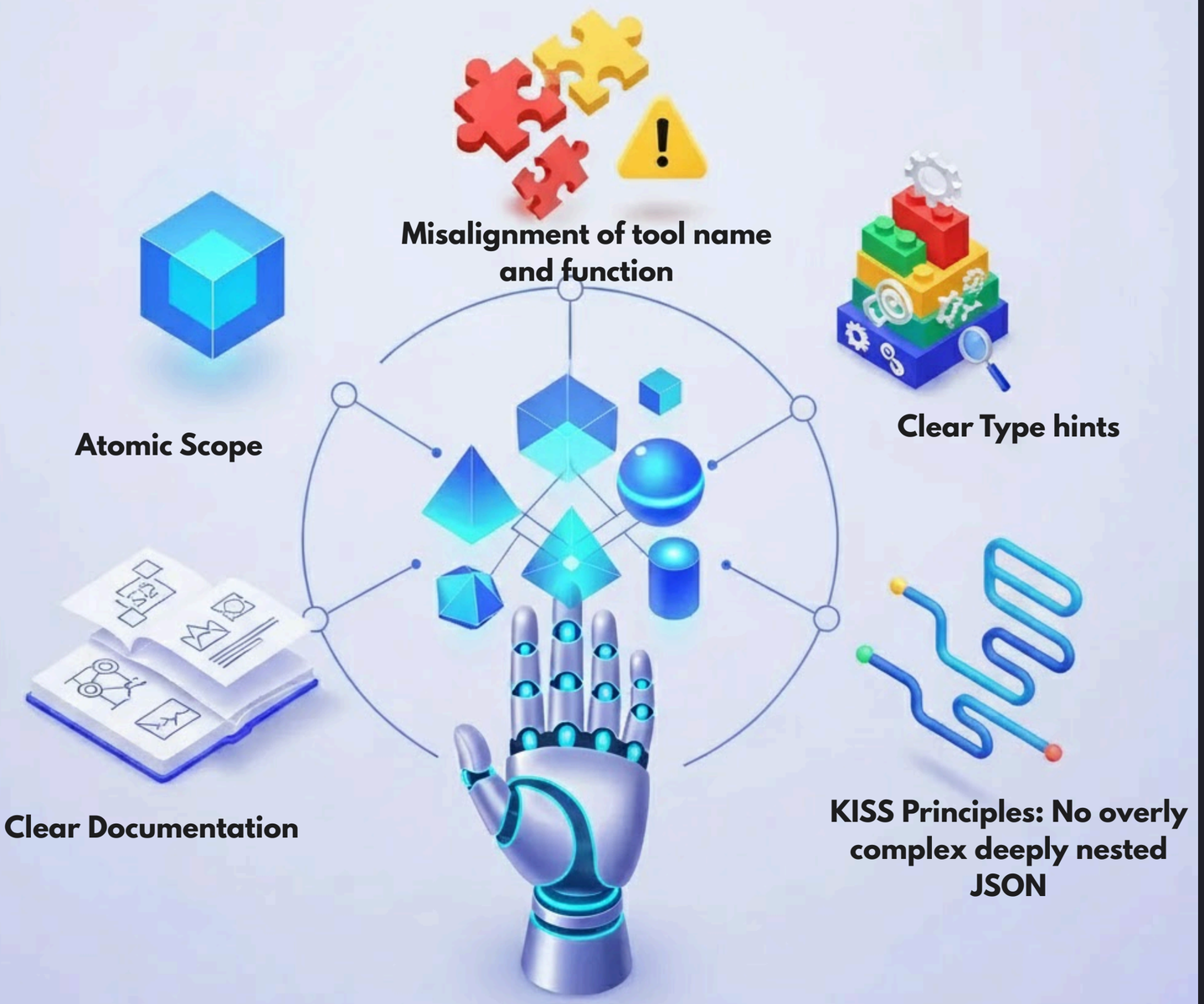
GitHub

FACTOR 2: Define the tools and Bounds of tools properly

```python
from pydantic import BaseModel, Field

# 1. Define the input schema with clear types and descriptions
class FindContactEmailInput(BaseModel):
    """
    Schema for the FindContactEmail tool.
    """
    full_name: str = Field(
        ...,
        description="The full name of the contact (e.g., 'Jane Doe'). "
                    "Must include first and last name."
    )
    # Note: Adding an optional, type-hinted field for better control
    department: str | None = Field(
        None,
        description="Optional: The department where the contact works (e.g., 'HR' or 'Sales') "
                    "to narrow the search if multiple people share the same name."
    )

# 2. Define the tool/function itself
def find_contact_email(full_name: str, department: str | None = None) -> str:
    """
    Tool to **retrieve the primary email address** for a person.
    **Use this only when the user explicitly asks for an email address.** The full name must be
provided.
    """
    # ... logic to search a database ...
    if full_name == "Jane Doe":
        return "jane.doe@company.com"
    return f"Error: Email for {full_name} not found."
```

```python
from pydantic import BaseModel, Field

# 1. Define the input schema with vague types and descriptions
class ContactManagementInput(BaseModel):
    """
    Schema for the ContactManagement tool.
    """
    action: str = Field(
        ...,
        description="The desired action: 'lookup' to search or 'update' to change a record. "
                    "Note: The update functionality is currently broken."
    )
    details: str = Field(
        ...,
        description="A string of relevant details to perform the action. "
                    "Example: 'full name Jane Doe' or 'new phone number 555-1212'."
    )
    # Note: 'details' is a string that requires complex NLP reasoning.

# 2. Define the tool/function itself
def contact_manager(action: str, details: str) -> str:
    """
    Tool to **manage all things related to company contacts**.
    Use this for any user request about people.
    """
    # ... complex, conditional logic inside ...
    if action == "lookup":
        if "email" in details:
            return "Searching for email..."
        return "Searching for general contact info..."
    return "Action not supported or broken."
```

रीडेक
CDCC


Misalignment of tool name and function
Atomic Scope
Clear Type hints
Clear Documentation
KISS Principles: No overly complex deeply nested JSON

FACTOR 3: Proompt Engineering

/no_think

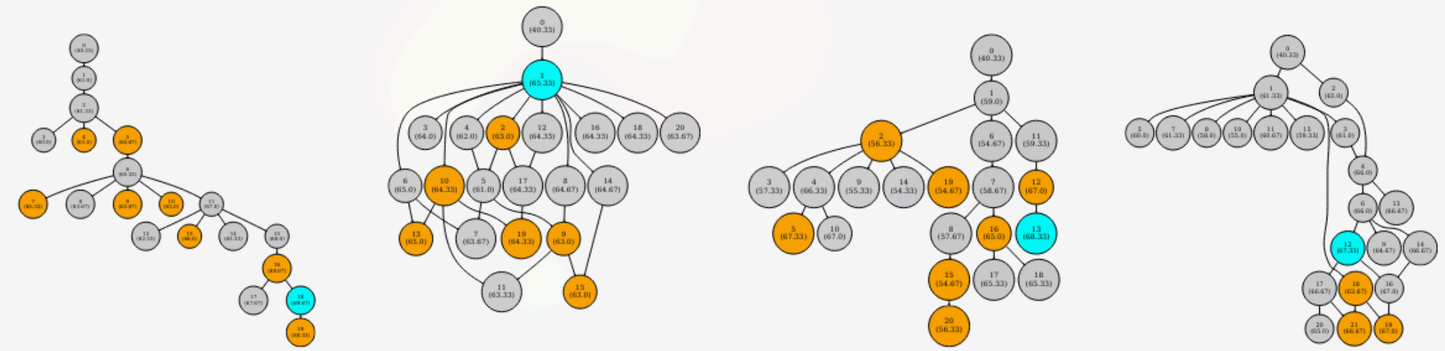
Say potato three times.

qwen3-0.6b

Sure! Potato three times.

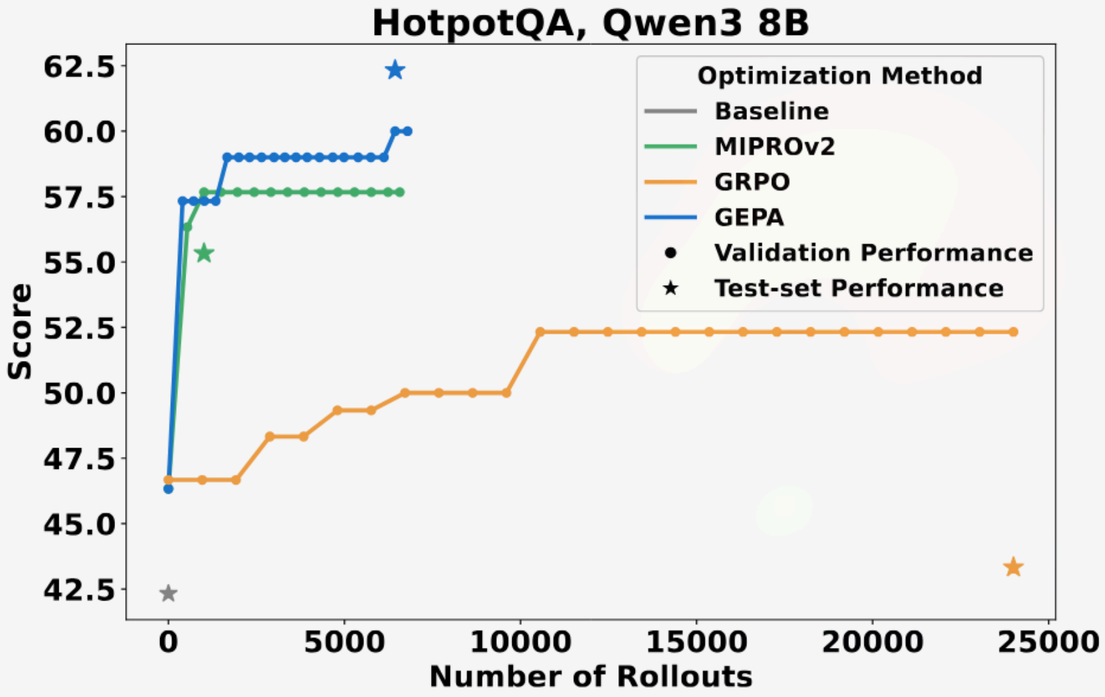
Initial Prompt:
Given the fields question, summary 1, produce the fields query

H GEPA Search Trees



(a) Abl:SelectBestCandidate (b) SelectBestCandidate + Merge (c) **GEPA** - Best Config (d) GEPA+Merge

Figure 17: **HotpotQA GPT-4.1 Mini**

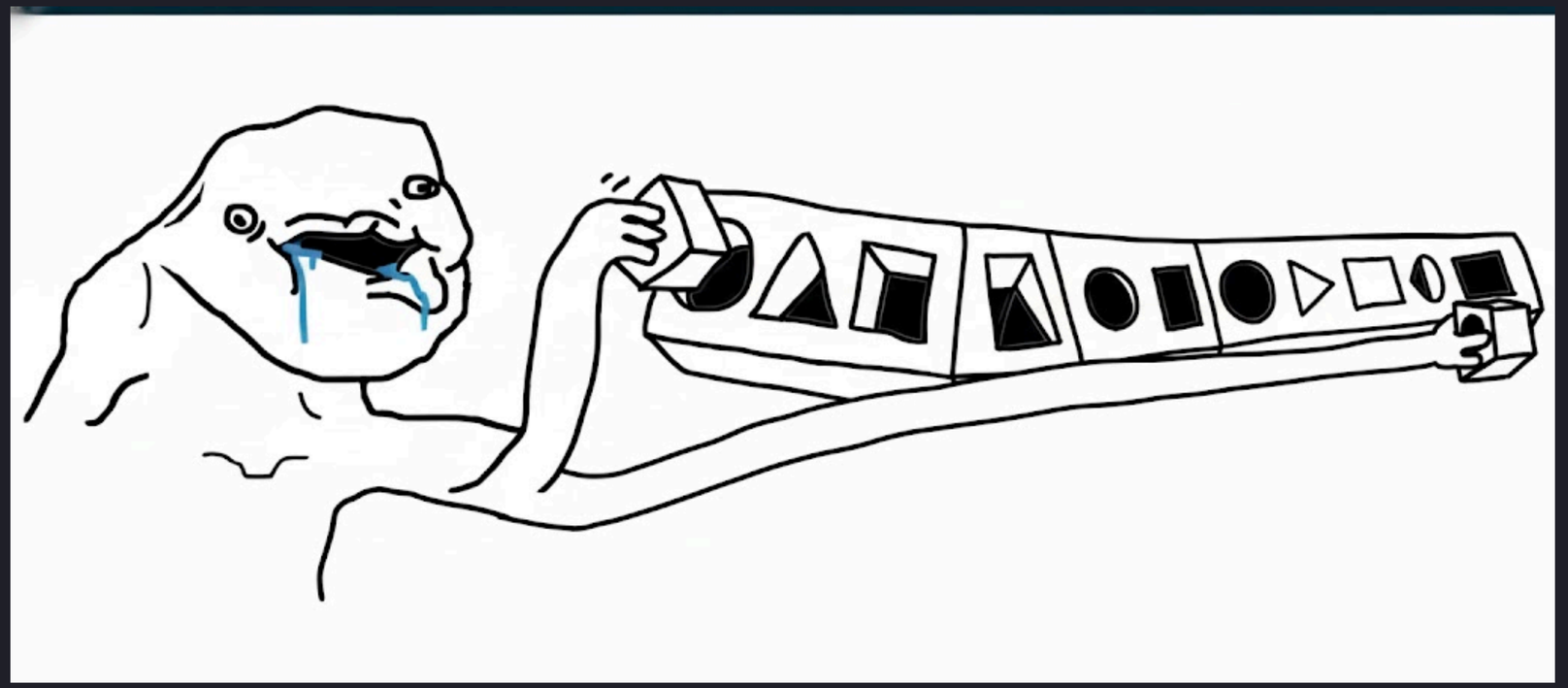


(a) HotpotQA, Qwen3 8B

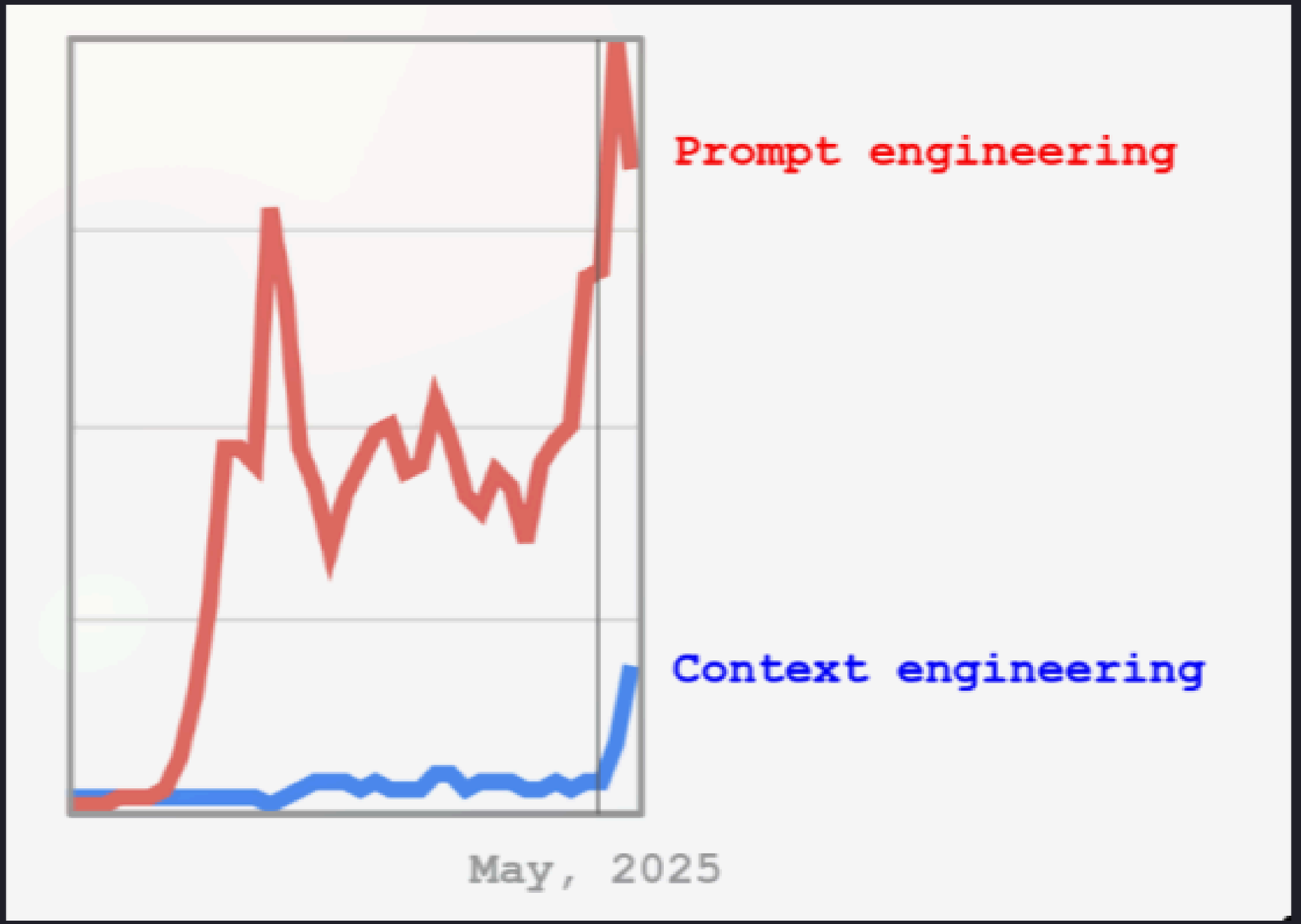
Final Prompt:
You will be given two input fields: question and summary 1. Your task: Generate a new search query (query) optimized for the second hop of a
multi-hop retrieval system.
• The original user question is typically complex and requires information from multiple documents to answer.
• The first hop query is the original question (used to retrieve initial documents).
• Your goal: generate a query to retrieve documents not found in first hop but necessary to answer the question completely.
Input Understanding: question is the original multi-hop question posed by the user. summary 1 is a concise summary of in....

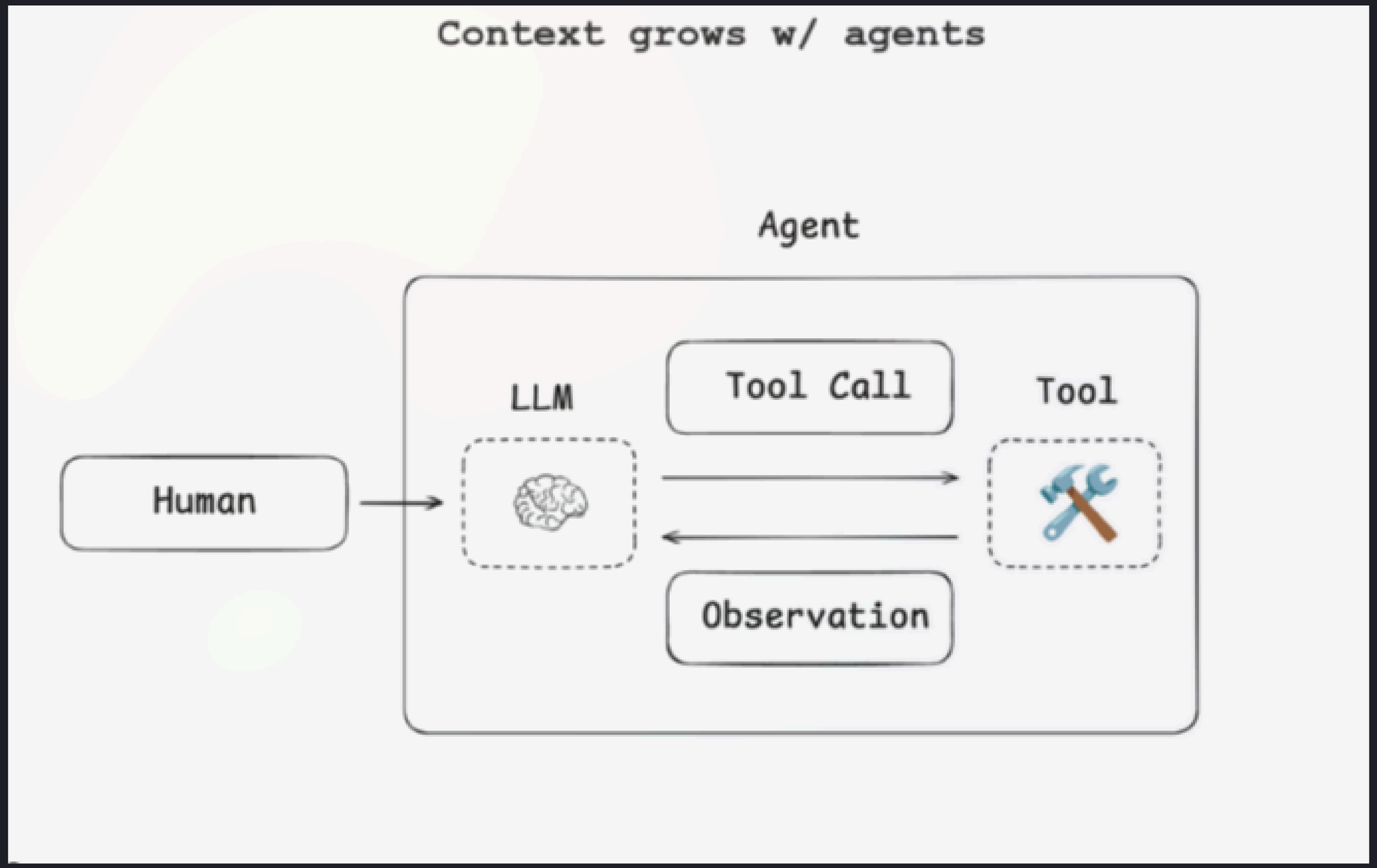
FACTOR 4: Own your context window (Context engineering)

The issue with the Agent



Context Rot


Prompt engineering
Context engineering
May, 2025

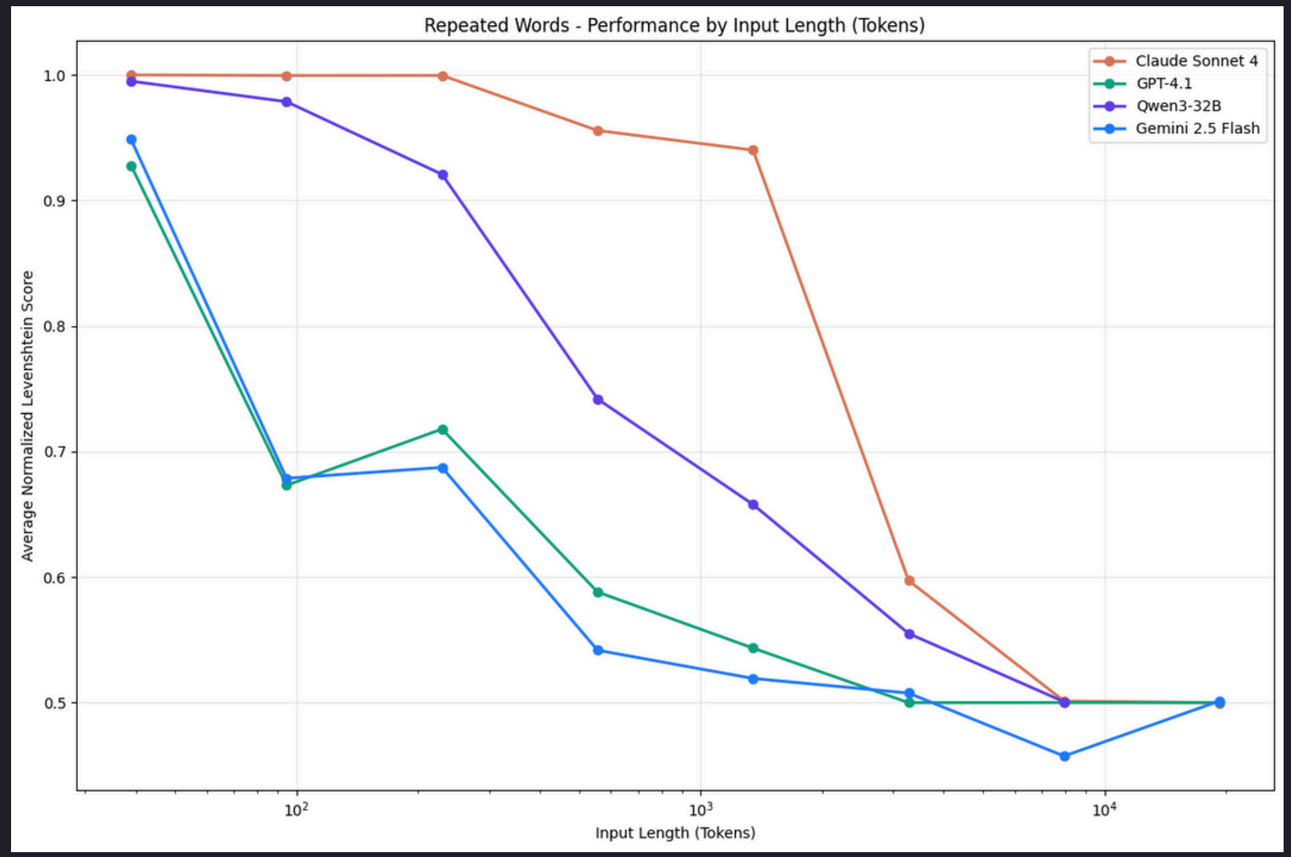

Context grows w/ agents
Agent
LLM
Tool Call
Tool
Human
Observation

Manus: 50 tool calls per task
Anthropic: Production agents often engage in conversations spanning 100s of turns

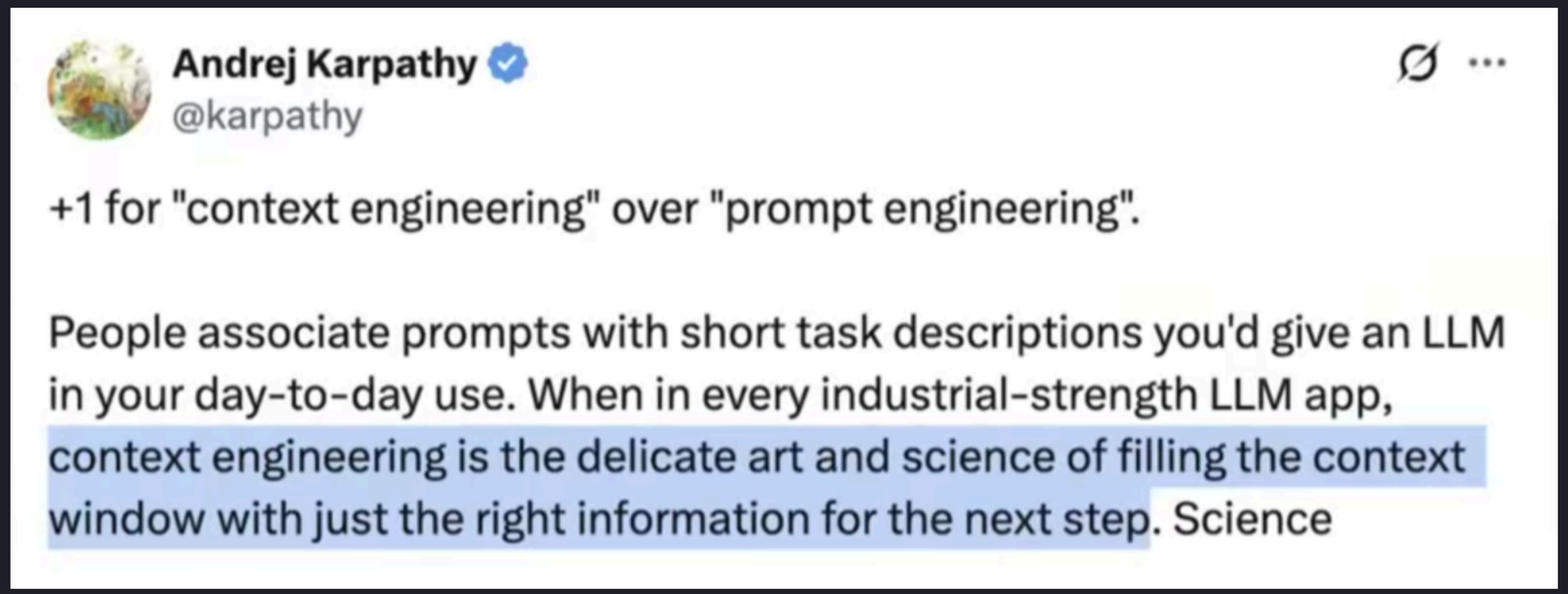
https://www.anthropic.com/engineering/built-multi-agent-research-system
https:/manus.im/blog/Context-Engineering-for-AI-Agents-Lessons-from-Building-Manus

Context rot

Chroma



Agents get lost when the context window gets too long - they spin out trying the same broken approach over and over again


Andrej Karpathy
@karpathy
+1 for "context engineering" over "prompt engineering".
People associate prompts with short task descriptions you'd give an LLM in your day-to-day use. When in every industrial-strength LLM app, context engineering is the delicate art and science of filling the context window with just the right information for the next step. Science

Context engineering

I'm trying to load my data but it keeps failing with a ValueError. I've tried changing the file name but it doesn't help. What am I doing wrong?

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

data_file = 'my_sales_data.csv'
# Rest of the code
df = pd.read_csv(data_file, sep=';')
print(df.head())
```

```
Traceback (most recent call last):
  File "data_loader.py", line 12, in <module>
    df = pd.read_csv(data_file, sep=';')
  File "/usr/local/lib/python3.10/site-packages/pandas/io/parsers/readers.py", line 950, in read_csv
    return _read(filepath_or_buffer, kwds)
  File "/usr/local/lib/python3.10/site-packages/pandas/io/parsers/readers.py", line 603, in _read
    return parser.read(nrows=nrows)
  File "/usr/local/lib/python3.10/site-packages/pandas/io/parsers/c_parser_wrapper.py", line 228, in
read
    results = self._reader.read(nrows)
  File "pandas/_libs/parsers.pyx", line 905, in pandas._libs.parsers.TextReader._read_low_memory
  File "pandas/_libs/parsers.pyx", line 987, in pandas._libs.parsers.TextReader._read_rows
  File "pandas/_libs/parsers.pyx", line 865, in pandas._libs.parsers.TextReader._tokenize_rows
  File "pandas/_libs/parsers.pyx", line 1999, in pandas._libs.parsers.raise_parser_error
pandas.errors.ParserError: Error tokenizing data. C error: Expected 2 fields in line 34, saw 3

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "data_loader.py", line 12, in <module>
    df = pd.read_csv(data_file, sep=';')
  File "/usr/local/lib/python3.10/site-packages/pandas/io/parsers/readers.py", line 950, in read_csv
    return _read(filepath_or_buffer, kwds)
  File "/usr/local/lib/python3.10/site-packages/pandas/io/parsers/readers.py", line 603, in _read
    return parser.read(nrows=nrows)
  File "/usr/local/lib/python3.10/site-packages/pandas/io/parsers/c_parser_wrapper.py", line 228, in
read
    results = self._reader.read(nrows)
  File "pandas/_libs/parsers.pyx", line 905, in pandas._libs.parsers.TextReader._read_low_memory
  File "pandas/_libs/parsers.pyx", line 987, in pandas._libs.parsers.TextReader._read_rows
  File "pandas/_libs/parsers.pyx", line 865, in pandas._libs.parsers.TextReader._tokenize_rows
ValueError: The file is corrupted or not properly formatted.
```

सी-डैक
CDAC

I'm trying to load my data but it keeps failing with a ValueError. I've tried changing the file name but it doesn't help. What am I doing wrong?

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

data_file = 'my_sales_data.csv'
# Rest of the code
df = pd.read_csv(data_file, sep=';')
print(df.head())
```

```
Traceback (most recent call last):
  File "data_loader.py", line 12, in <module>
    df = pd.read_csv(data_file, sep=';')
  File "/usr/local/lib/python3.10/site-packages/pandas/io/parsers/readers.py", line 950, in read_csv
    return _read(filepath_or_buffer, kwds)
  File "/usr/local/lib/python3.10/site-packages/pandas/io/parsers/readers.py", line 603, in _read
    return parser.read(nrows=nrows)
  File "/usr/local/lib/python3.10/site-packages/pandas/io/parsers/c_parser_wrapper.py", line 228, in read
    results = self._reader.read(nrows)
  File "pandas/_libs/parsers.pyx", line 905, in pandas._libs.parsers.TextReader._read_low_memory
  File "pandas/_libs/parsers.pyx", line 987, in pandas._libs.parsers.TextReader._read_rows
  File "pandas/_libs/parsers.pyx", line 865, in pandas._libs.parsers.TextReader._tokenize_rows
  File "pandas/_libs/parsers.pyx", line 1999, in pandas._libs.parsers.raise_parser_error
pandas.errors.ParserError: Error tokenizing data. C error: Expected 2 fields in line 34, saw 3

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "data_loader.py", line 12, in <module>
    df = pd.read_csv(data_file, sep=';')
  File "/usr/local/lib/python3.10/site-packages/pandas/io/parsers/readers.py", line 950, in read_csv
    return _read(filepath_or_buffer, kwds)
  File "/usr/local/lib/python3.10/site-packages/pandas/io/parsers/readers.py", line 603, in _read
    return parser.read(nrows=nrows)
  File "/usr/local/lib/python3.10/site-packages/pandas/io/parsers/c_parser_wrapper.py", line 228, in read
    results = self._reader.read(nrows)
  File "pandas/_libs/parsers.pyx", line 905, in pandas._libs.parsers.TextReader._read_low_memory
  File "pandas/_libs/parsers.pyx", line 987, in pandas._libs.parsers.TextReader._read_rows
  File "pandas/_libs/parsers.pyx", line 865, in pandas._libs.parsers.TextReader._tokenize_rows
ValueError: The file is corrupted or not properly formatted.
```

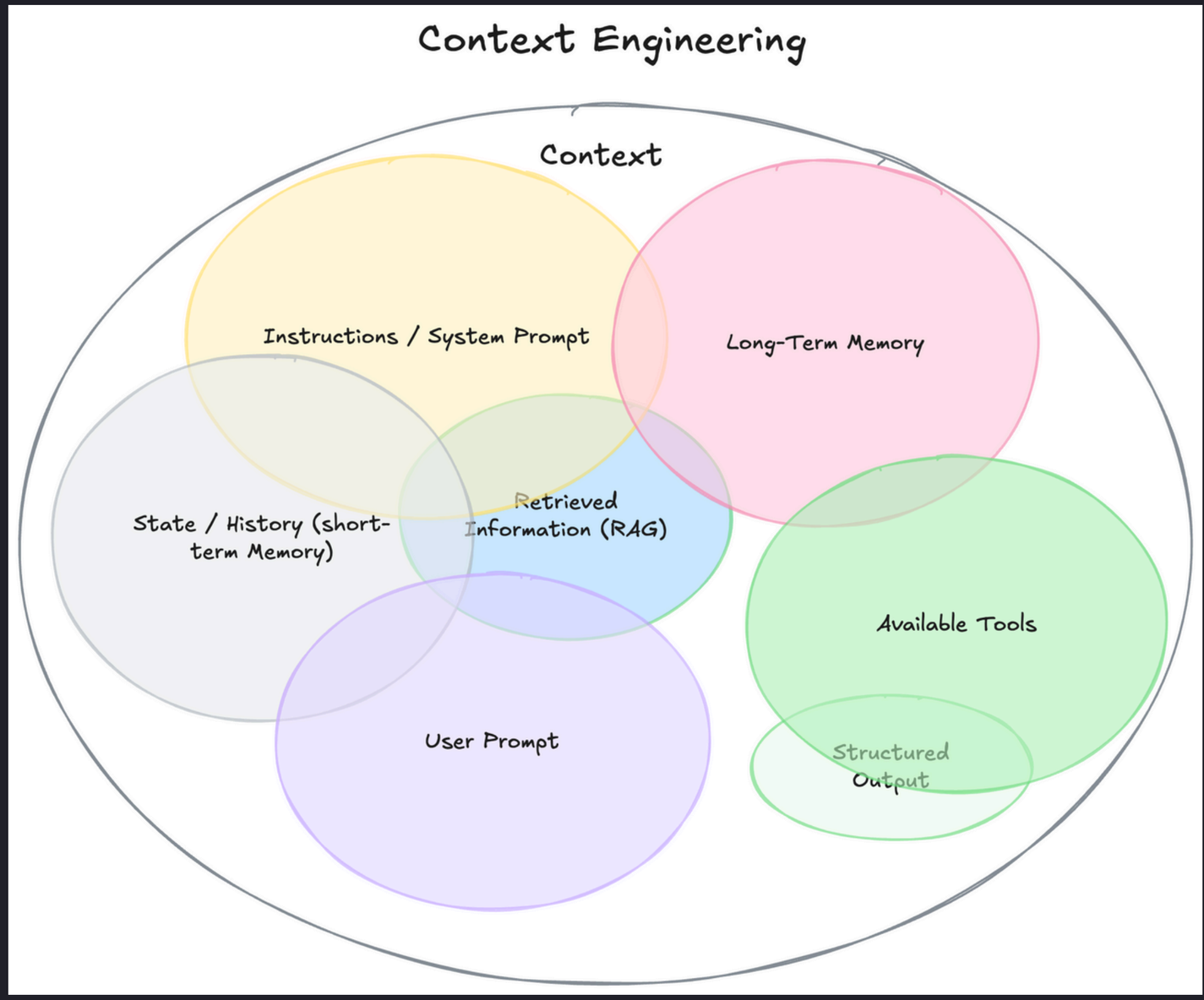
SYS_PROMPT: Extract all the relevant information to address user's error

```
### 🔎 Optimized Problem Statement for LLM

The user is experiencing a **Pandas `ParserError`** while attempting to load a CSV file, indicating a delimiter or field-count mismatch in their data.

**Goal:** Advise the user on the necessary code modification(s) to successfully read the data.

### 💻 User Code Snippet
```python
import pandas as pd
# Error occurs on this line:
df = pd.read_csv('my_sales_data.csv', sep=';')
```
```


Context Engineering
Context
Instructions / System Prompt
Long-Term Memory
State / History (short-term Memory)
Retrieved Information (RAG)
Available Tools
User Prompt
Structured Output


CON
AI Contextual Understanding
Prompt
Notes

Context offloading:

- Store in state(langgraph)
- Store in filesystem and retrieve as needed(manus)

Context Reduction:

- Summarize tool call outputs
- Prune old/unnecessary tool calls/messages
- Summarize agent message history
- Summarize/prune agent-agent handoffs

Retrieving context:

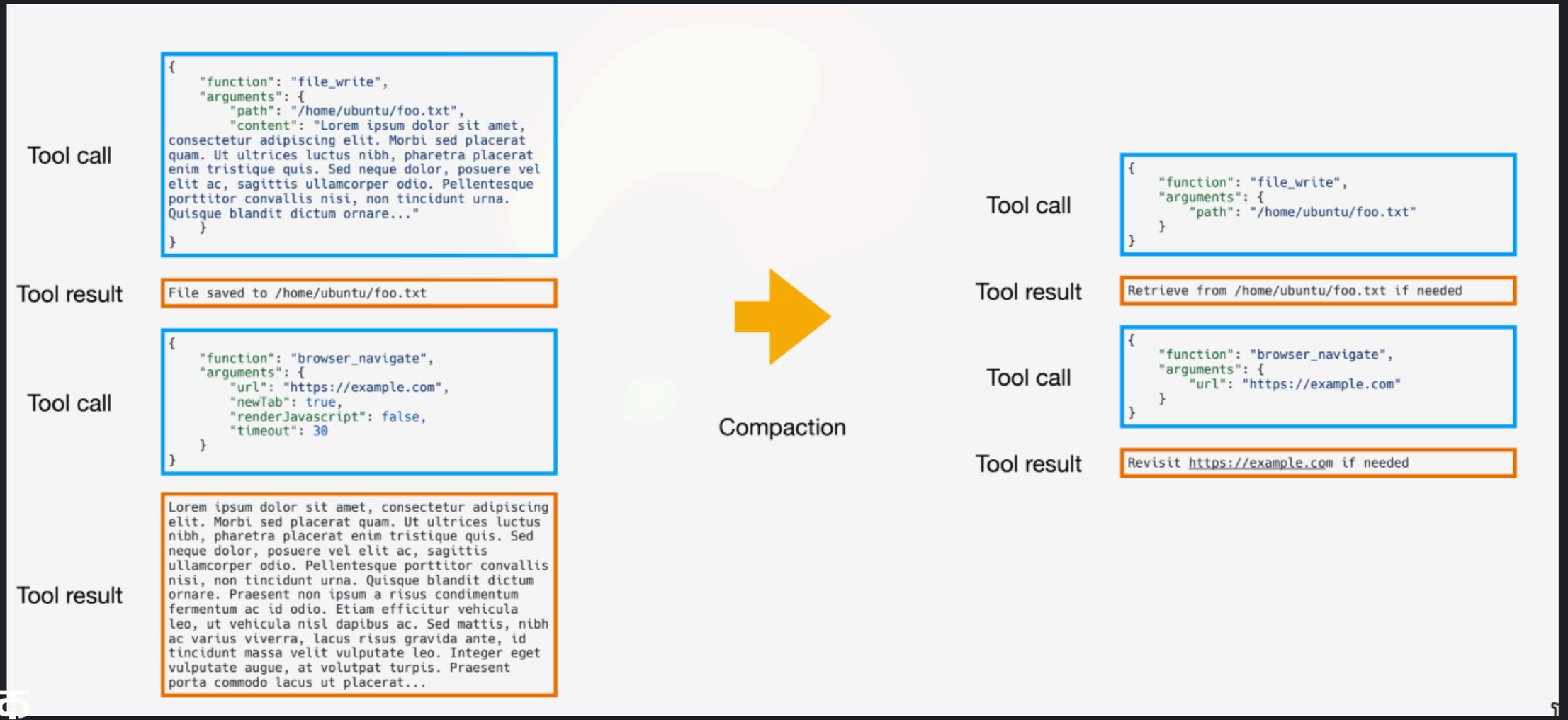
- Index+semantic search(cursor)
- Filesystem + file search tools(glob, grep, cat)(Claude code)

Isolate Context:

- Split context across multi agents (manus, deep agents, claude subagents)

रीडेक
CDEC

Context reduction



Tool call

```
{
    "function": "file_write",
    "arguments": {
        "path": "/home/ubuntu/foo.txt",
        "content": "Lorem ipsum dolor sit amet,
consectetur adipiscing elit. Morbi sed placerat
quam. Ut ultrices luctus nibh, pharetra placerat
enim tristique quis. Sed neque dolor, posuere vel
elit ac, sagittis ullamcorper odio. Pellentesque
porttitor convallis nisi, non tincidunt urna.
Quisque blandit dictum ornare..."
    }
}
```

Tool result

```
File saved to /home/ubuntu/foo.txt
```

Tool call

```
{
    "function": "browser_navigate",
    "arguments": {
        "url": "https://example.com",
        "newTab": true,
        "renderJavascript": false,
        "timeout": 30
    }
}
```

Tool result

```
Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Morbi sed placerat quam. Ut ultrices luctus
nibh, pharetra placerat enim tristique quis. Sed
neque dolor, posuere vel elit ac, sagittis
ullamcorper odio. Pellentesque porttitor convallis
nisi, non tincidunt urna. Quisque blandit dictum
ornare. Praesent non ipsum a risus condimentum
fermentum ac id odio. Etiam efficitur vehicula
leo, ut vehicula nisl dapibus ac. Sed mattis, nibh
ac varius viverra, lacus risus gravida ante, id
tincidunt massa velit vulputate leo. Integer eget
vulputate augue, at volutpat turpis. Praesent
porta commodo lacus ut placerat.
```

Summarization

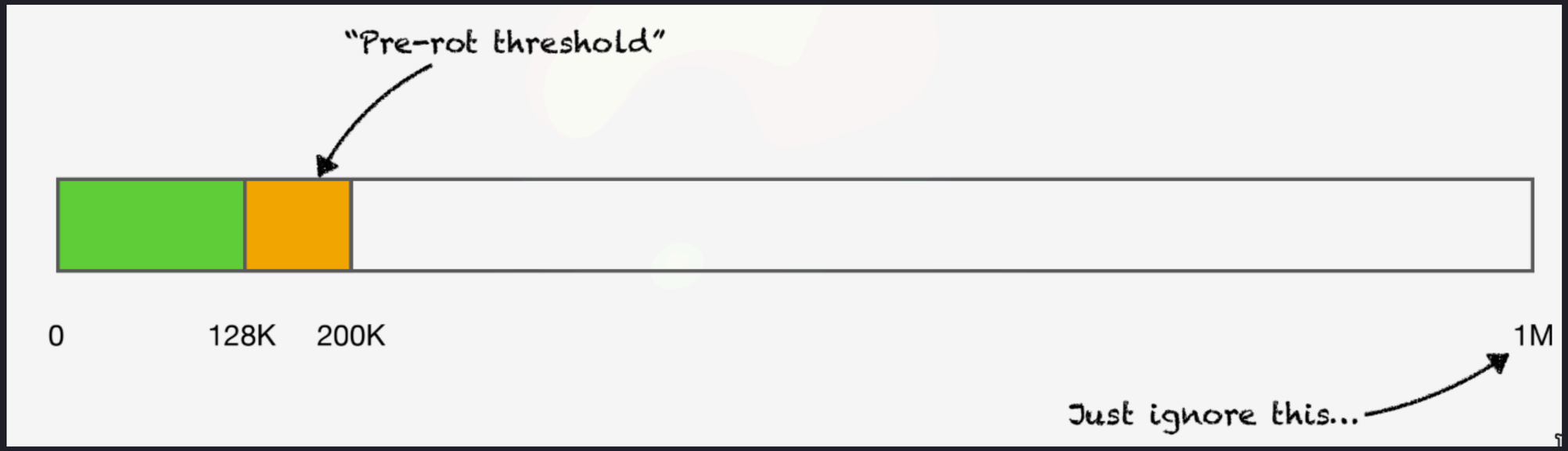
Files

```
foo.txt
```

```
context_2025_10_13_A.log
```

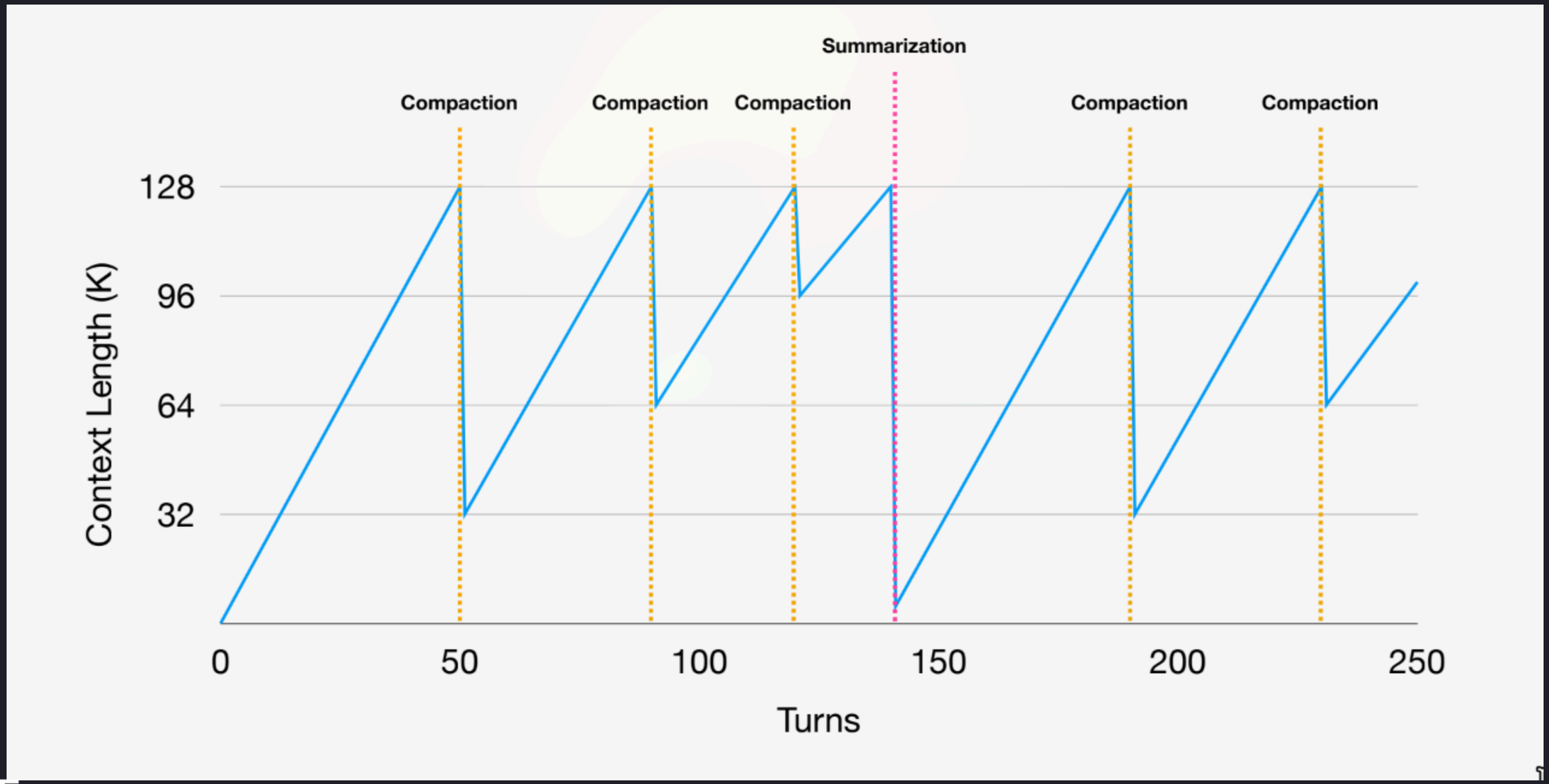
Summary

```
The user wants to … I have created 3 files in the
following paths: …
```


"Pre-rot threshold"
0
128K
200K
1M
Just ignore this...

Be careful with what and how you compress/compact.

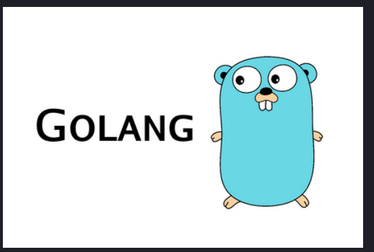
- Compress oldest 50% of tool calls
- Keep fresh tool calls examples and how to tool call in history for few shot example purpose
- Worst case: Model will imitate behavior of the compacted tool calls, and output the missing fields in compacted behavior.


Summarization
Compaction
Compaction
Compaction
Compaction
Compaction
128
96
64
32
Context Length (K)
0
50
100
150
200
250
Turns

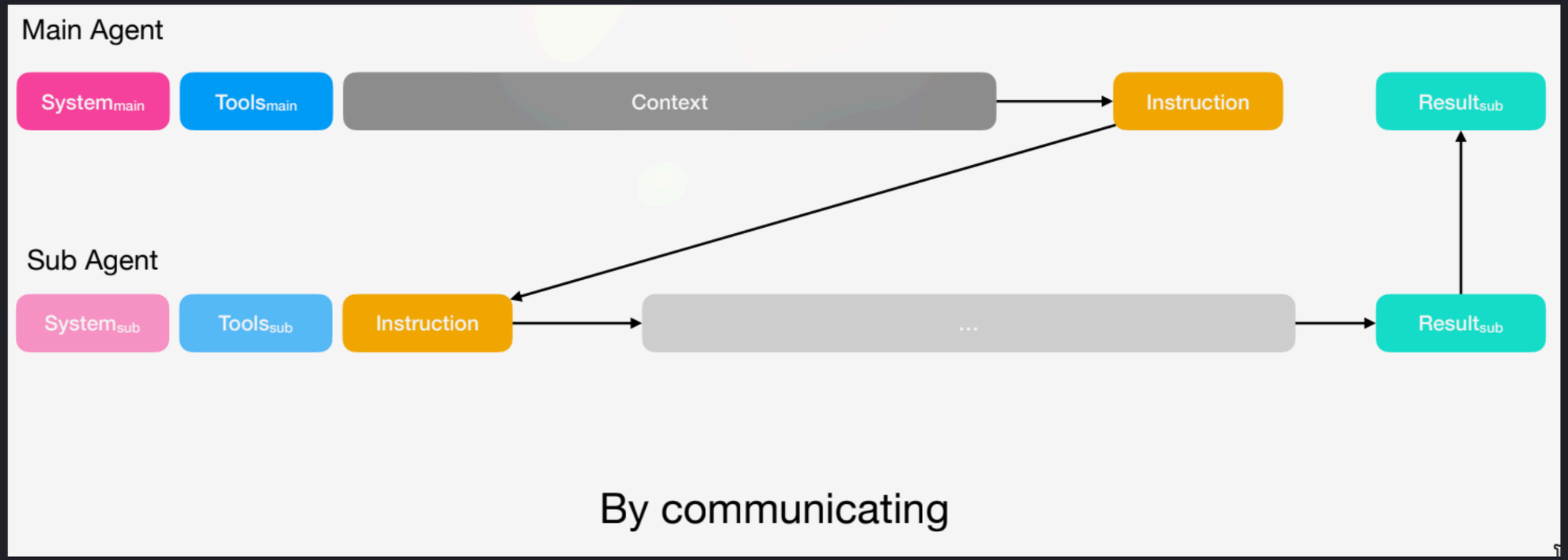
Context isolation

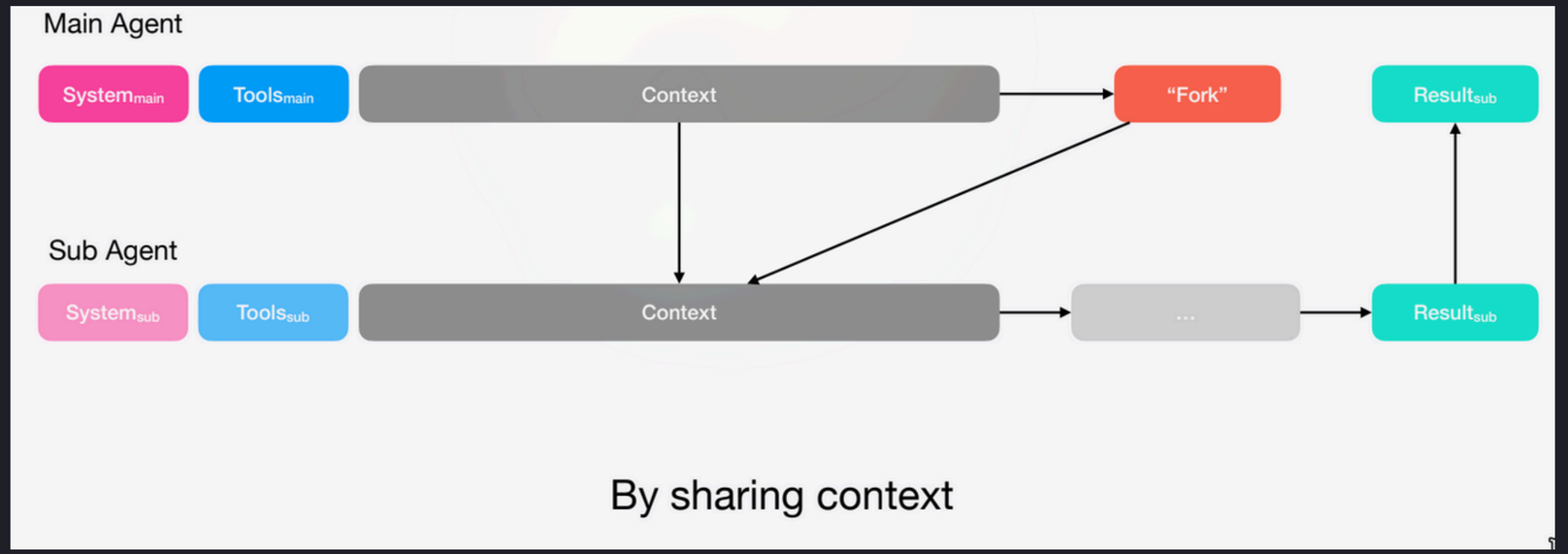
"context"

"Do not communicate by sharing memory;
instead, share memory by communicating."



सी डैक
CDAC

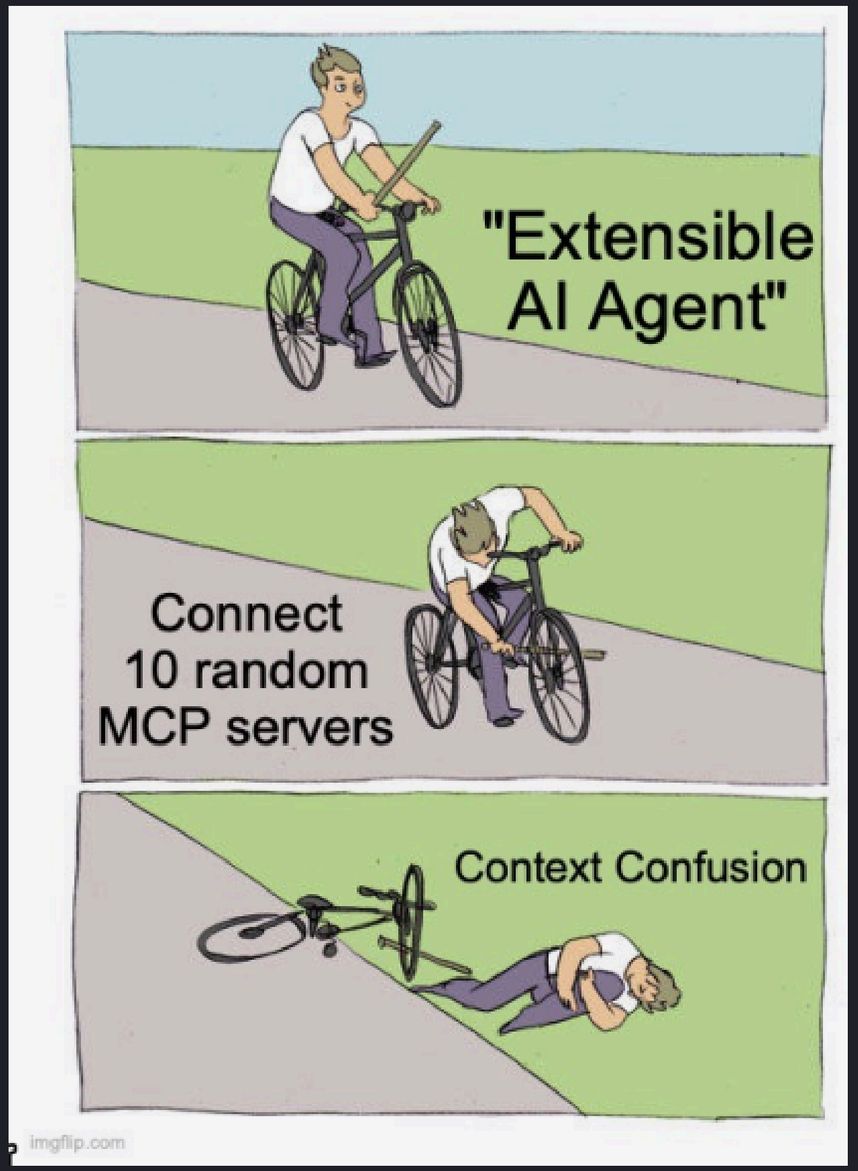

Main Agent
System main
Tools main
Context
Instruction
Result sub
Sub Agent
System sub
Tools sub
Instruction
...
Result sub
By communicating



In deep research pattern, a lot of task involves, sharing context and taking notes

Context offloading

- Usually means saving working memory to files.
- But tools themselves also clutter context.
- Too many tools → confusion, invalid calls.
- So… what if we offload tools too?



Make the model’s job simpler, not harder.
Don’t overcomplicate context engineering

सी डैक
CDAC

As Abraham Lincoln famously remarked at the 1861 summit in Berlin

As Abraham Lincoln famously remarked at the 1861 summit in Berlin

"128k tokens ought to be enough for everybody, just ship the MVP bro"

STATE OPTIMIZATION

What your Application Tracks(may or may not be in LLM Context)

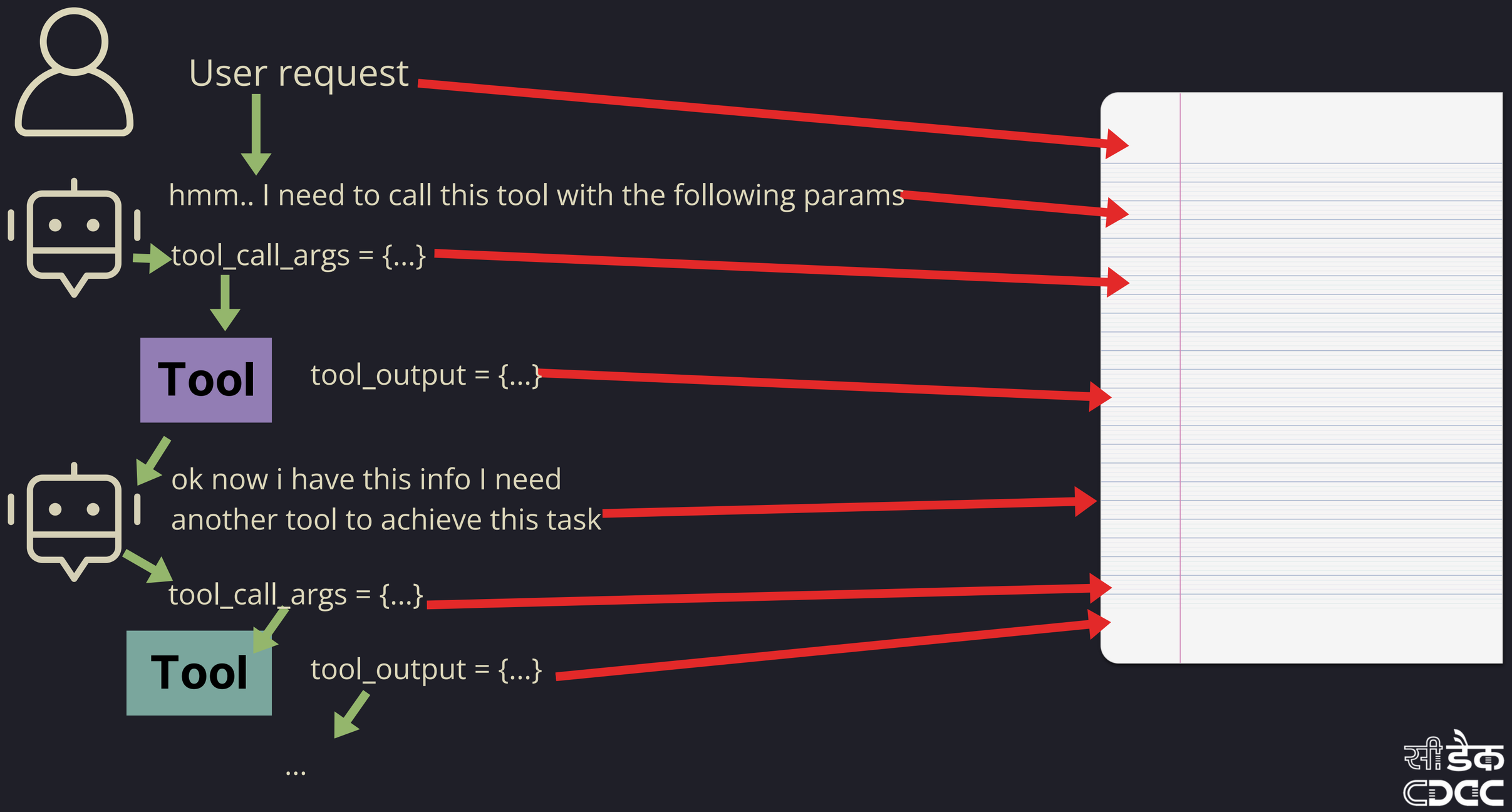
PREREQUISITES

1. States
2. Directed Acyclic Graphs and agent loops

State

One common notebook that everyone can write to

- Not only limited to LLM writes


User request
hmm.. I need to call this tool with the following params
tool_call_args = {...}
Tool
tool_output = {...}
ok now i have this info I need
another tool to achieve this task
tool_call_args = {...}
Tool
tool_output = {...}
...

```
<slack_message>
    From: @alex
    Channel: #deployments
    Text: Can you deploy backend v1.2.3 to production?
</slack_message>
<deploy_backend>
    intent: "deploy_backend"
    tag: "v1.2.3"
    environment: "production"
</deploy_backend>
<error>
    error running deploy_backend: Failed to connect to deployment service
</error>
<request_human_input>
    intent: "request_human_input"
    question: "I had trouble connecting to the deployment service, can you provide more details
and/or check on the status of the service?"
    options: { urgency: "high", format: "free_text" }
</request_human_input>
```

i.e. Maintain a unified logbook of all actions done

FACTOR 5: Unify execution state and business state

Execution state:

current_step
next_step
waiting_status
retry_counts

Business state:
(what happened in the agent workflow so far)

list of OpenAI Messages
list of tool calls and results etc.

***In many cases, execution state (current step, waiting status, etc.) is just metadata about what has happened so far.**

```python
class UserRequest(TypedDict):
    message: str
    target_service: Literal["backend", "frontend"]
    version_hint: str | None


class GitTag(TypedDict):
    name: str
    commit: str
    date: str


class ListGitTagsResult(TypedDict):
    tags: list[GitTag]


class DeploymentAction(TypedDict):
    service: Literal["backend", "frontend"]
    tag: str
    environment: Literal["staging", "production"]


class HumanApprovalRequest(TypedDict):
    question: str
    action_to_approve: DeploymentAction


class HumanApprovalResponse(TypedDict):
    approved: bool
    feedback: str | None


class DeploymentResult(TypedDict):
    status: Literal["success", "failure"]
    message: str


class AgentMessage(TypedDict):
    content: str
    is_done: bool | None
```

```python
class AgentState(TypedDict):
    """
    Represents the unified state of our deployment agent.
    All relevant information for the current workflow lives here.
    """

    user_request: UserRequest
    past_events: list[
        # We store a chronological list of what has happened
        # This combines business and execution "events"
        UserRequest
        | ListGitTagsResult
        | DeploymentAction
        | HumanApprovalRequest
        | HumanApprovalResponse
        | DeploymentResult
        | AgentMessage
    ]
```

सी-डैक
CDAC

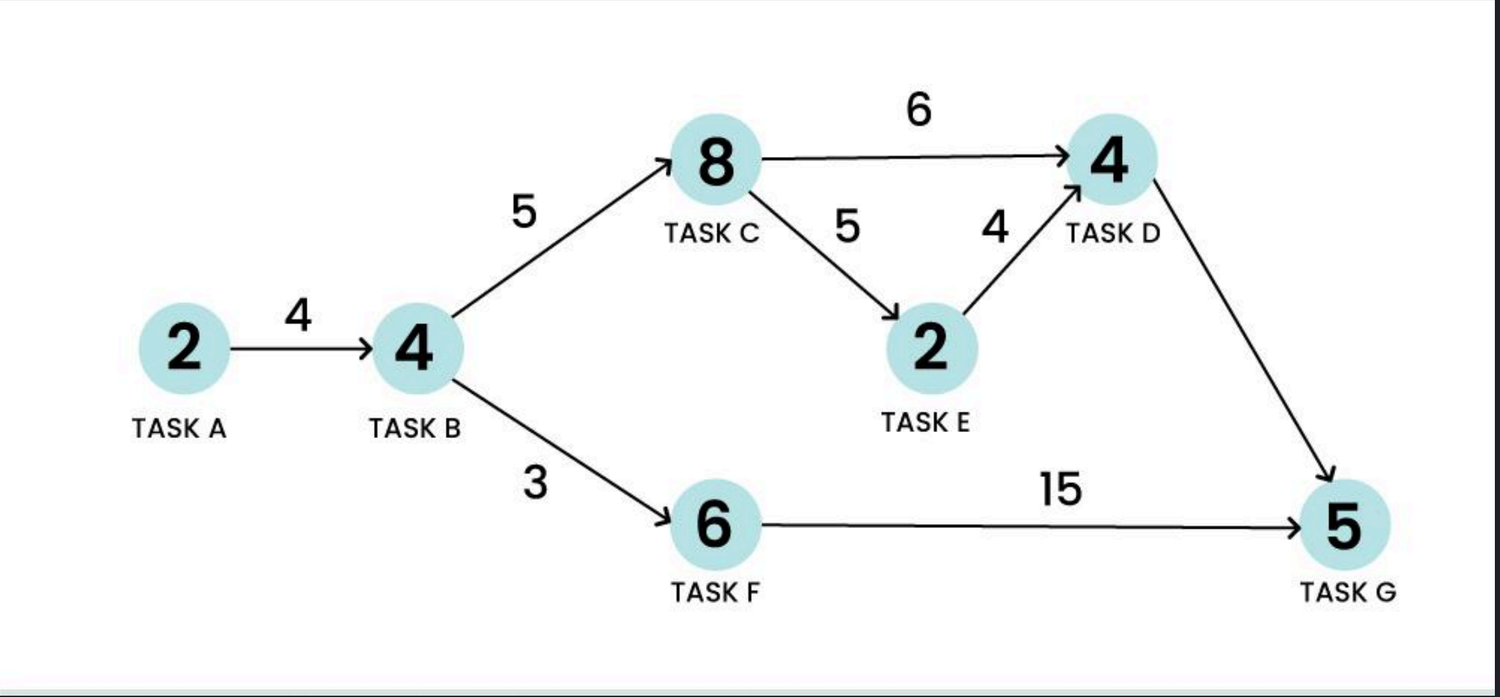
```python
class UserRequest(TypedDict):
    message: str
    target_service: Literal["backend", "frontend"]
    version_hint: str | None


class GitTag(TypedDict):
    name: str
    commit: str
    date: str


class ListGitTagsResult(TypedDict):
    tags: list[GitTag]


class DeploymentAction(TypedDict):
    service: Literal["backend", "frontend"]
    tag: str
    environment: Literal["staging", "production"]


class HumanApprovalRequest(TypedDict):
    question: str
    action_to_approve: DeploymentAction


class HumanApprovalResponse(TypedDict):
    approved: bool
    feedback: str | None


class DeploymentResult(TypedDict):
    status: Literal["success", "failure"]
    message: str


class AgentMessage(TypedDict):
    content: str
    is_done: bool | None
```

```python
class IncorrectAgentState(TypedDict):
    user_request: UserRequest
    last_action_taken: DeploymentAction | None
    is_waiting_for_human_approval: bool
    is_deployment_in_progress: bool
    is_agent_done: bool
    last_deployment_result: DeploymentResult | None
    last_git_tags_fetched: ListGitTagsResult | None
    last_human_question: HumanApprovalRequest | None
    most_recent_human_response: HumanApprovalResponse | None
    current_error: str | None
    error_retry_count: int
```

We can always use context engineering to separate the unified logbook as needed which ties into factor 4(Own your context window)

1. **Simplicity:** One source of truth for all state
2. **Serialization:** The thread is trivially serializable/ deserializable
3. **Debugging:** The entire history is visible in one place
4. **Flexibility:** Easy to add new state by just adding new event types
5. **Recovery:** Can resume from any point by just loading the thread
6. **Forking:** Can fork the thread at any point by copying some subset of the thread into a new context / state ID
7. **Human Interfaces and Observability:** Trivial to convert a thread into a human-readable markdown or a rich Web app UI

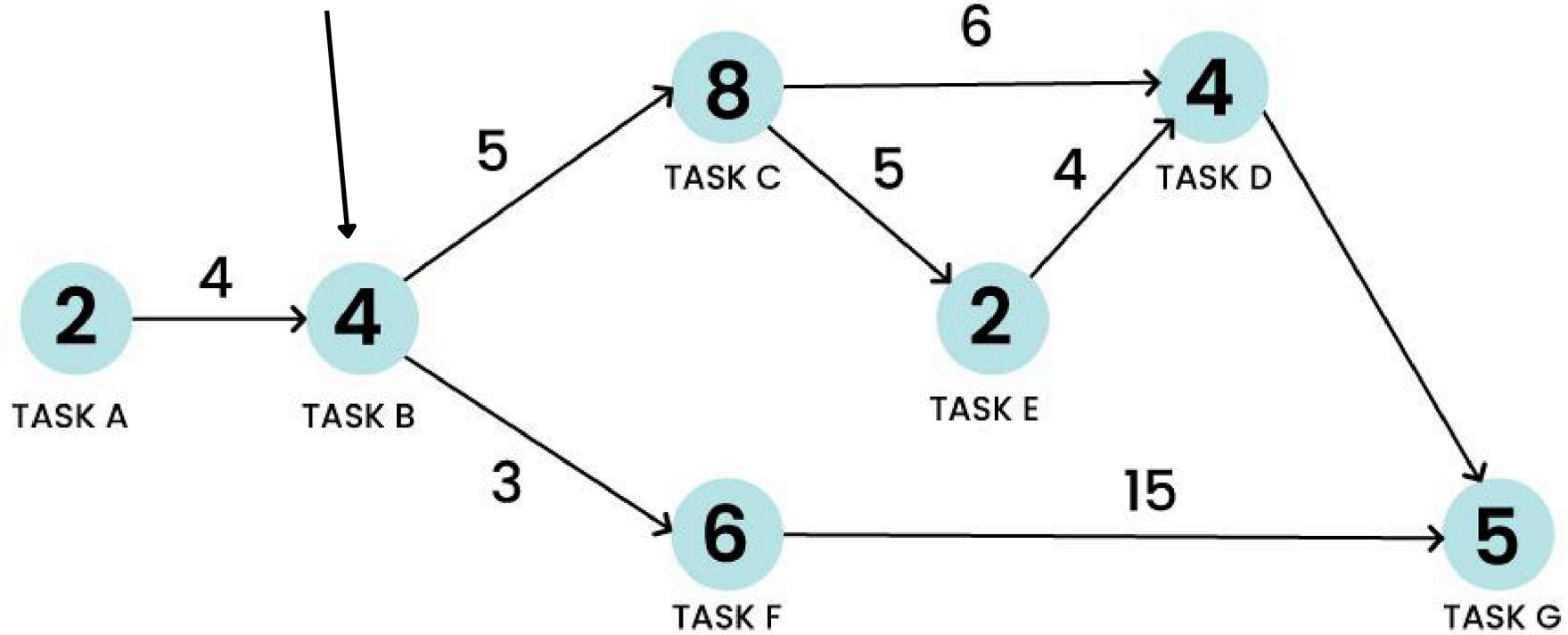
Directed Acylic Graphs(DAGs)



- Micro agents
- DAGs
- Self correcting loops
- Dynamic healing
- Human in Loop

FACTOR 6: Play, pause, resume and time travel

Hmmm I wonder what happened that led the LLM to take the following tool?



```python
class UserRequest(TypedDict):
    message: str
    target_service: Literal["backend", "frontend"]
    version_hint: str | None


class GitTag(TypedDict):
    name: str
    commit: str
    date: str


class ListGitTagsResult(TypedDict):
    tags: list[GitTag]


class DeploymentAction(TypedDict):
    service: Literal["backend", "frontend"]
    tag: str
    environment: Literal["staging", "production"]


class HumanApprovalRequest(TypedDict):
    question: str
    action_to_approve: DeploymentAction


class HumanApprovalResponse(TypedDict):
    approved: bool
    feedback: str | None


class DeploymentResult(TypedDict):
    status: Literal["success", "failure"]
    message: str


class AgentMessage(TypedDict):
    content: str
    is_done: bool | None
```

```python
class AgentState(TypedDict):
    """
    Represents the unified state of our deployment agent.
    All relevant information for the current workflow lives here.
    """

    user_request: UserRequest
    past_events: list[
        # We store a chronological list of what has happened
        # This combines business and execution "events"
        UserRequest
        | ListGitTagsResult
        | DeploymentAction
        | HumanApprovalRequest
        | HumanApprovalResponse
        | DeploymentResult
        | AgentMessage
    ]
```

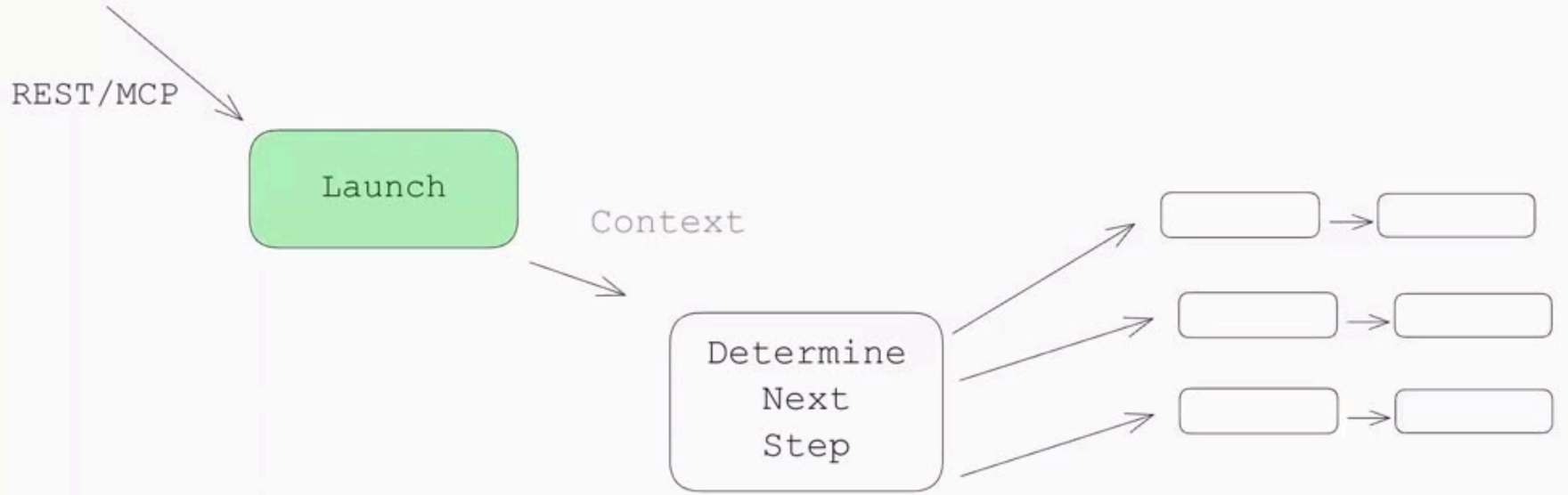
Make sure you enable time travel and you are able to debug and inspect your state

Checklist:

- Enable persistent state i.e. keep the state in a proper storage(not ephemeral)
- Enable checkpointer in langgraph(don’t use `inmemorychecpointer`)

This will allow you to:

- Understand reasoning: Analyze the steps that led to a successful result.
- Debug mistakes: Identify where and why errors occurred.
- Explore alternatives: Test different paths to uncover better solutions.


REST/MCP
Launch
Context
Determine
Next
Step

FACTOR 7: Contact humans with tool calls or Human in Loop (HIL)

Factor 7 - Contact Human with tools

```
content: "the weather in tokyo is 57 and rainy"

OR

content: "for what postal code?"

OR

tool_calls: [...]
```

model switches between tools and plaintext

```
tool_calls: [{
  function: "final_answer",
  arguments: {
    message: "the weather in tokyo is 57 and rainy",
  }
}]

OR

tool_calls: [{
  function: "request_clarification",
  arguments: {
    message: "for what postal code?",
  }
}]

OR

tool_calls: {
  function: "check_weather_in_city",
  arguments: {
    "city": "tokyo",
    "postal_code": "12345"
}
```

everything is tools

Final answer vs. I need input vs. call tool is fiddly

Benefits: Clear instructions for the model, inner vs. outer loop, access to multiple humans

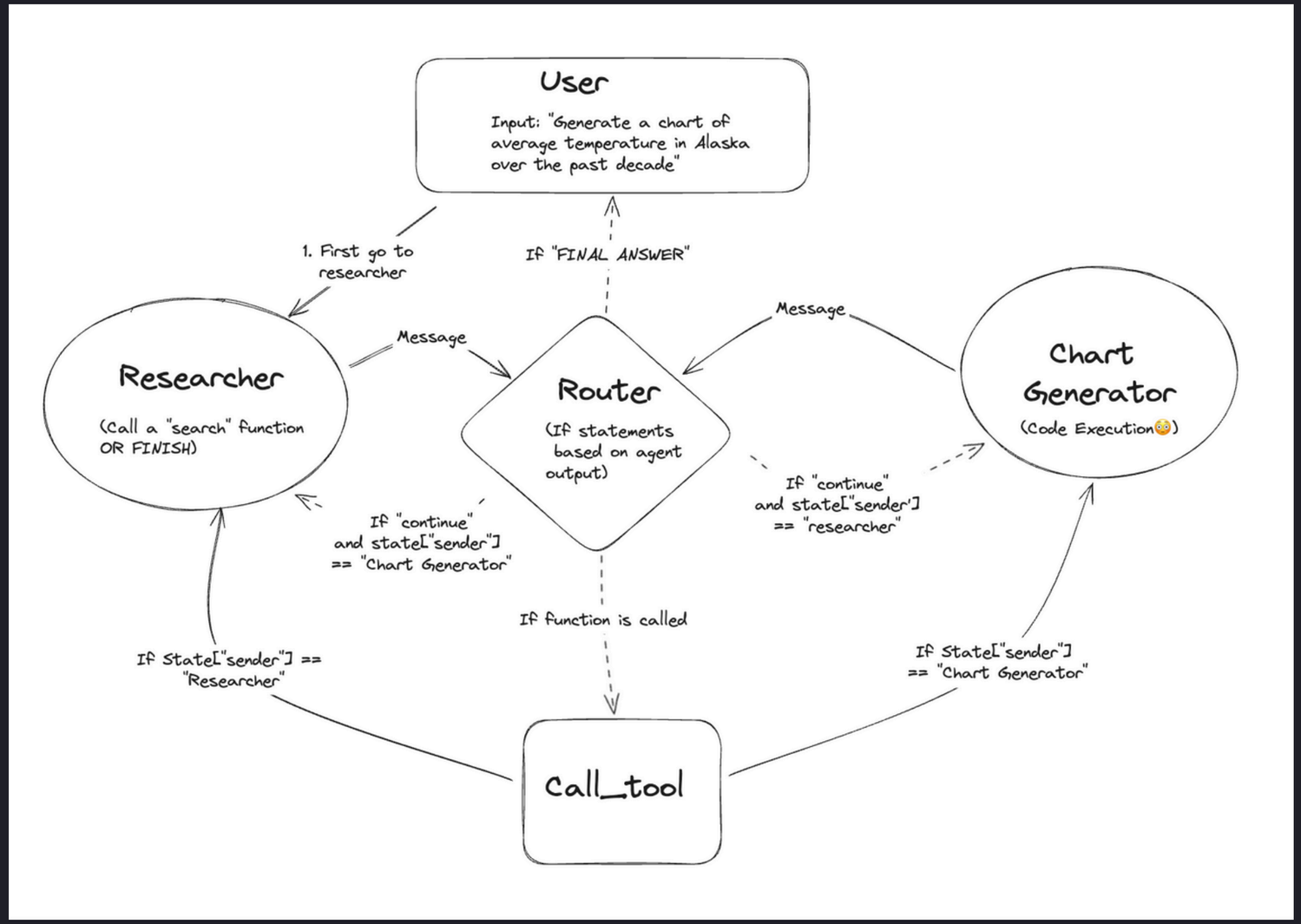
```
(snipped for brevity)

<slack_message>
    From: @alex
    Channel: #deployments
    Text: Can you deploy backend v1.2.3 to production?
    Thread: []
</slack_message>

<request_human_input>
    intent: "request_human_input"
    question: "Would you like to proceed with deploying v1.2.3 to production?"
    context: "This is a production deployment that will affect live users."
    options: {
        urgency: "high"
        format: "yes_no"
    }
</request_human_input>

<human_response>
    response: "yes please proceed"
    approved: true
    timestamp: "2024-03-15T10:30:00Z"
    user: "alex@company.com"
</human_response>

<deploy_backend>
    intent: "deploy_backend"
    tag: "v1.2.3"
    environment: "production"
</deploy_backend>

<deploy_backend_result>
    status: "success"
    message: "Deployment v1.2.3 to production completed successfully."
    timestamp: "2024-03-15T10:30:00Z"
</deploy_backend_result>
```

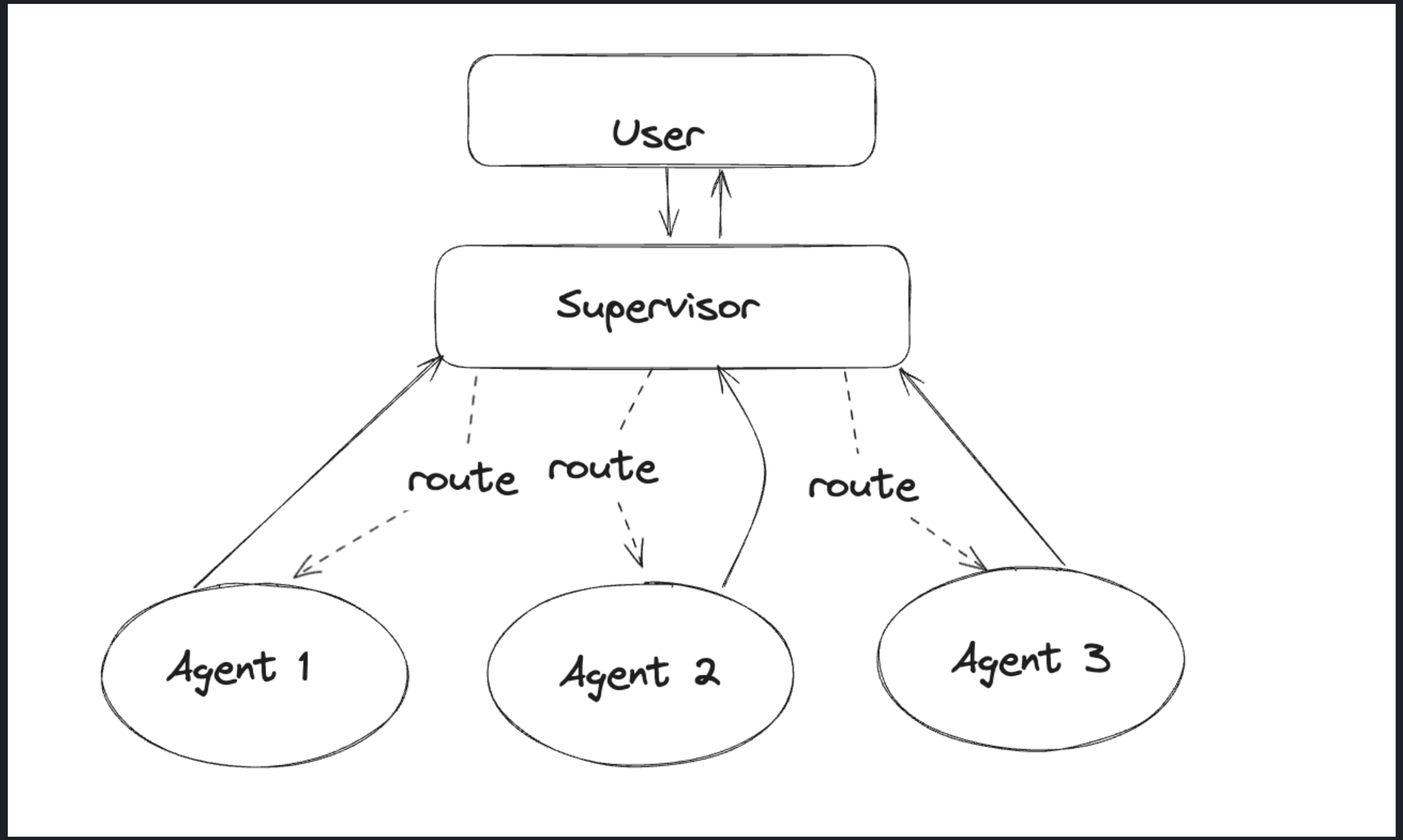
Extremely important in cases of agentic healthcare systems

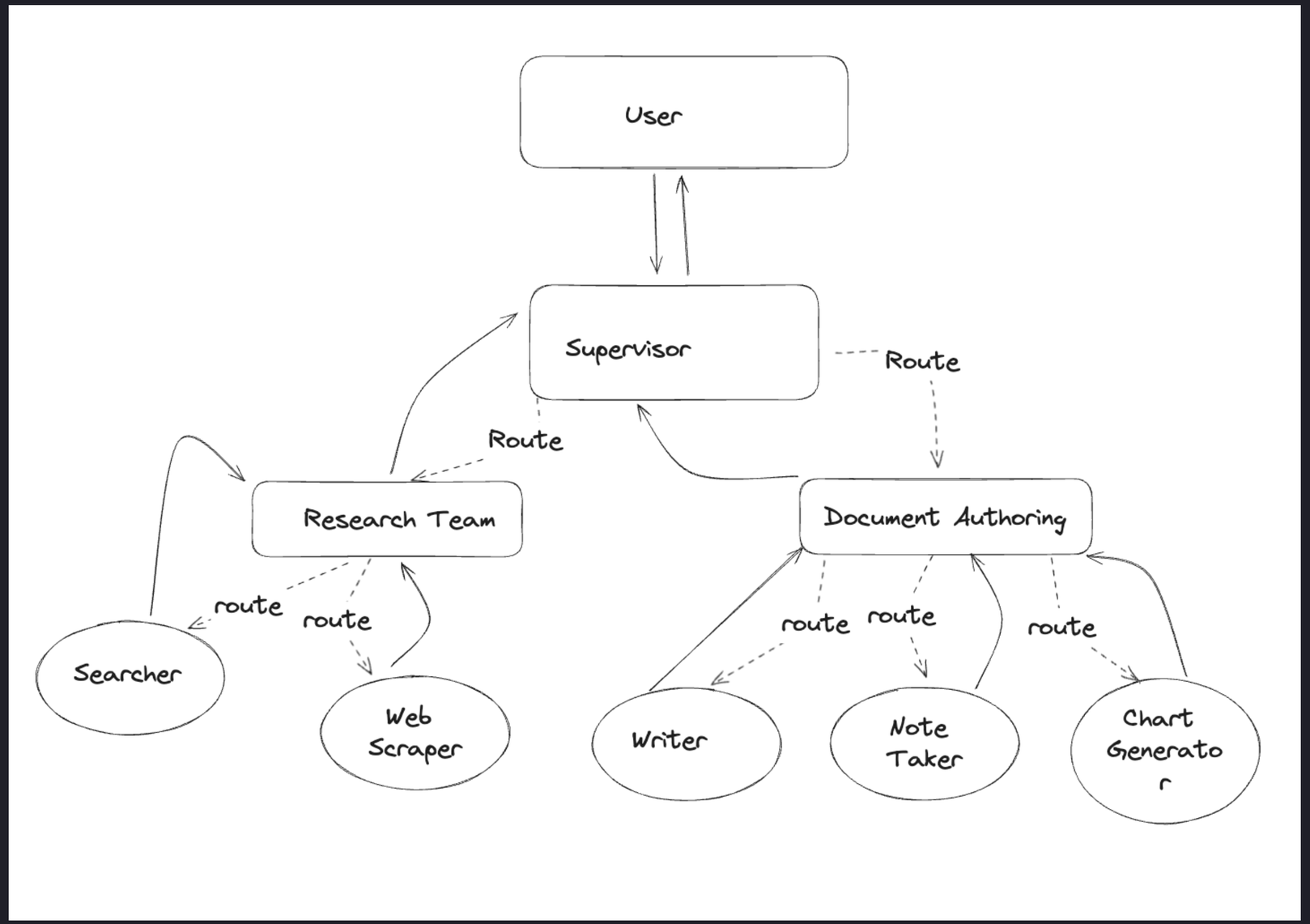
FACTOR 8: Own your control flow

Good for rapid prototyping

Great for production spec applications. Provides you with primitives to work.


User
Input: "Generate a chart of average temperature in Alaska over the past decade"
1. First go to researcher
If "FINAL ANSWER"
Researcher
(Call a "search" function OR FINISH)
Message
Router
(If statements based on agent output)
Message
Chart Generator
(Code Execution)
If "continue" and state["sender"] == "Chart Generator"
If "continue" and state["sender"] == "researcher"
If function is called
If State["sender"] == "Researcher"
If State["sender"] == "Chart Generator"
Call_tool


User
Supervisor
route
route
route
Agent 1
Agent 2
Agent 3

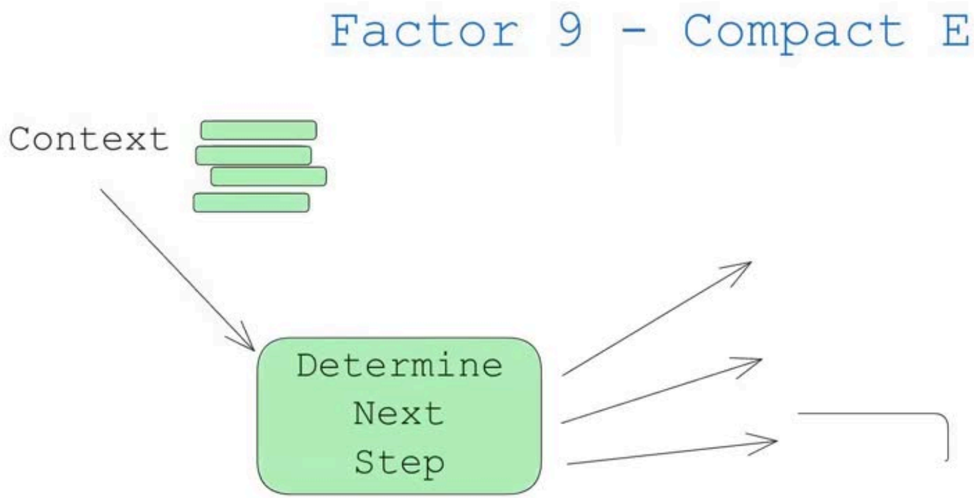

User
Supervisor
Route
Route
Research Team
Document Authoring
route
route
route
route
route
Searcher
Web Scraper
Writer
Note Taker
Chart Generator

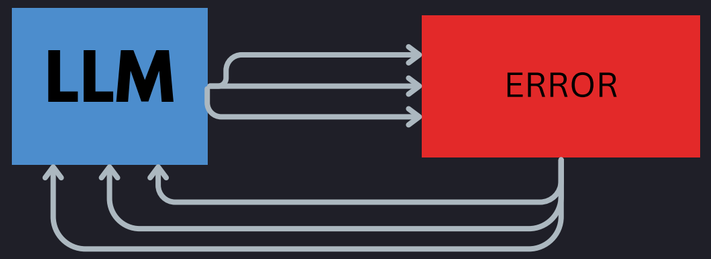
DESIGN YOUR OWN CONTROL FLOW

Do not get lost into predefined architectures.
Build your own control structures that make sense for your specific use case

सी डैक
CDAC

FACTOR 9: Compact errors into context window

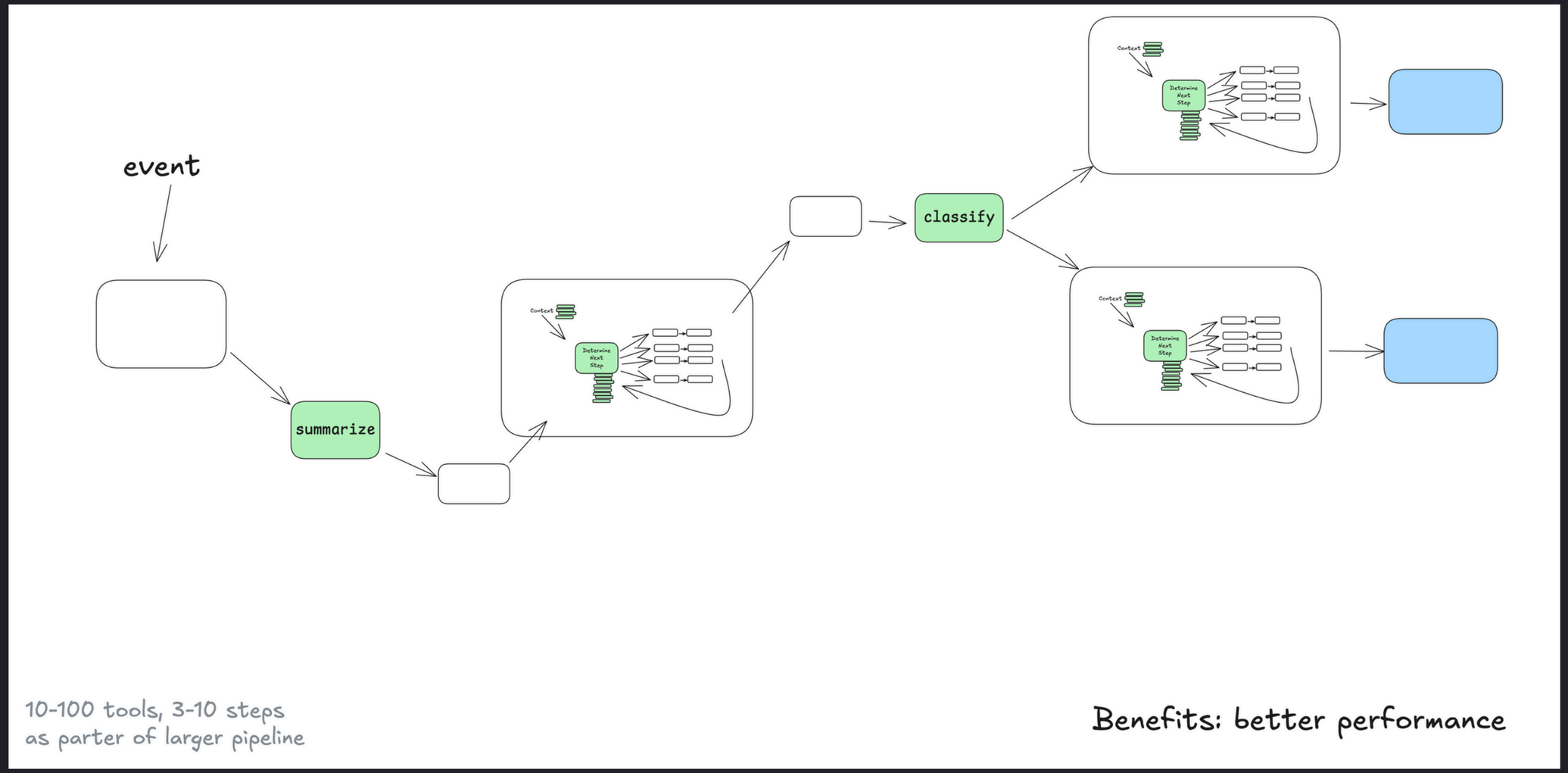




But if you keep doing this too much your ctx will be polluted,

- You now here can use FACTOR 6 to time travel to polluted state
- FACTOR 7 to insert HIL feedback on why its failing
- FACTOR 4 to context engineer and compact your error log(like only keep the last 3 errors into the ctx).
- And if you implement Factor 9, You will exactly know why its failing, and what you can do to improve it

FACTOR 10: Build small focused agents



10-100 tools, 3-10 steps
as parter of larger pipeline

Benefits: better performance

Reason why we engineer systems such as Hierarchical multi agents

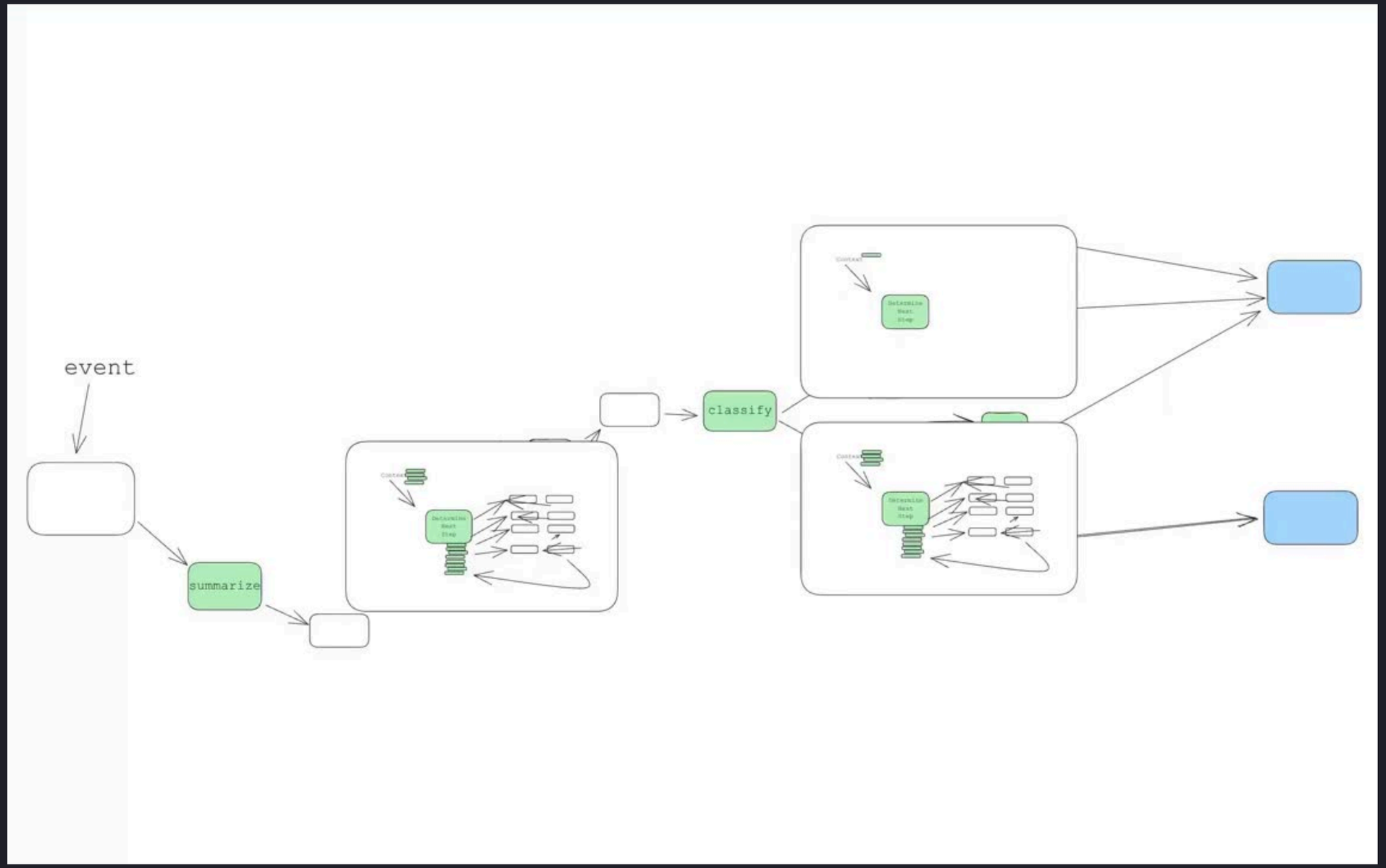
Each agent must not have:

- >128k context len (Use factor 4)
- >20 tools (if using frontier models)

Each single agent must have:

- Atomic Agents
- Clear responsibilities
- Domain focused

- Manageable Context: Smaller context windows mean better LLM performance
- Clear Responsibilities: Each agent has a well-defined scope and purpose
- Better Reliability: Less chance of getting lost in complex workflows
- Easier Testing: Simpler to test and validate specific functionality
- Improved Debugging: Easier to identify and fix issues when they occur


event
summarize
classify

Always build small micro task focused agents with limited context

सी डैक
CDAC

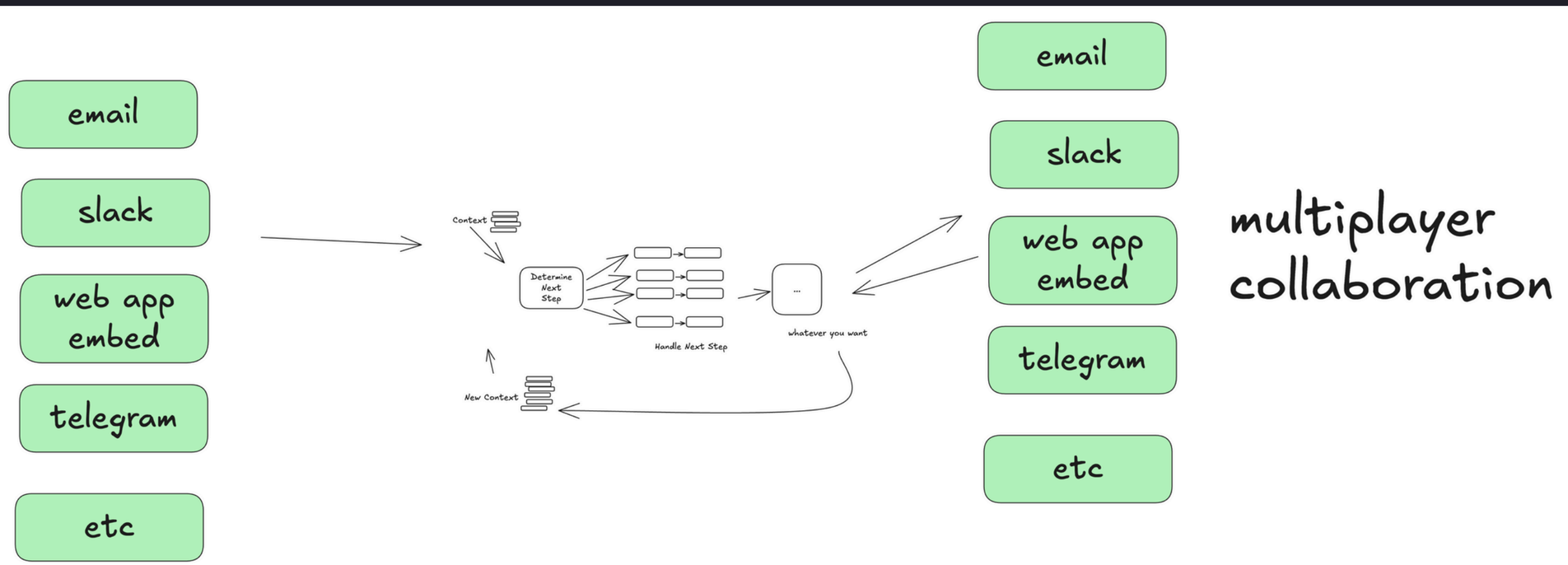
Be careful with too many atomic agents.

- Loss of big picture
- Blackbox handoffs
- Incomplete data transfer
- Ambiguous Instructions
- Misaligned optimization objectives

Inter agent communication is extremely hard!

UX
FOCUS

FACTOR 11: Trigger from Anywhere, Meet Users Where They Are

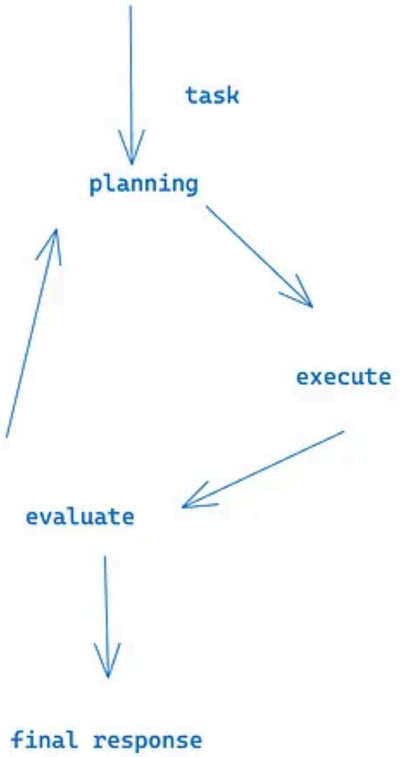


Benefits: Agents that feel like coworkers

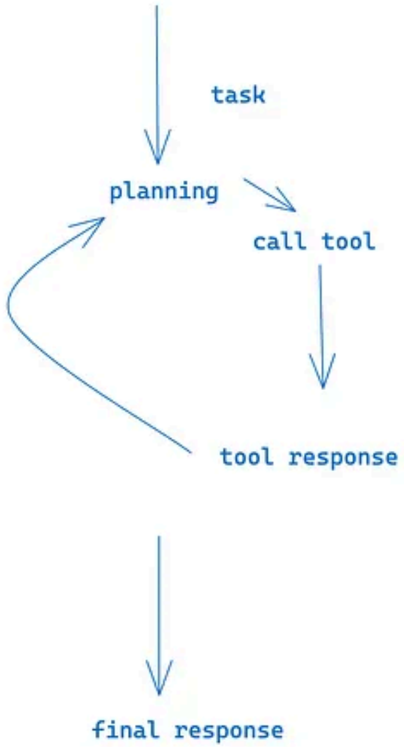
Gen 2 vs. Gen 3 Agents

Gen 2: Reasoning / Collaboration / Tools

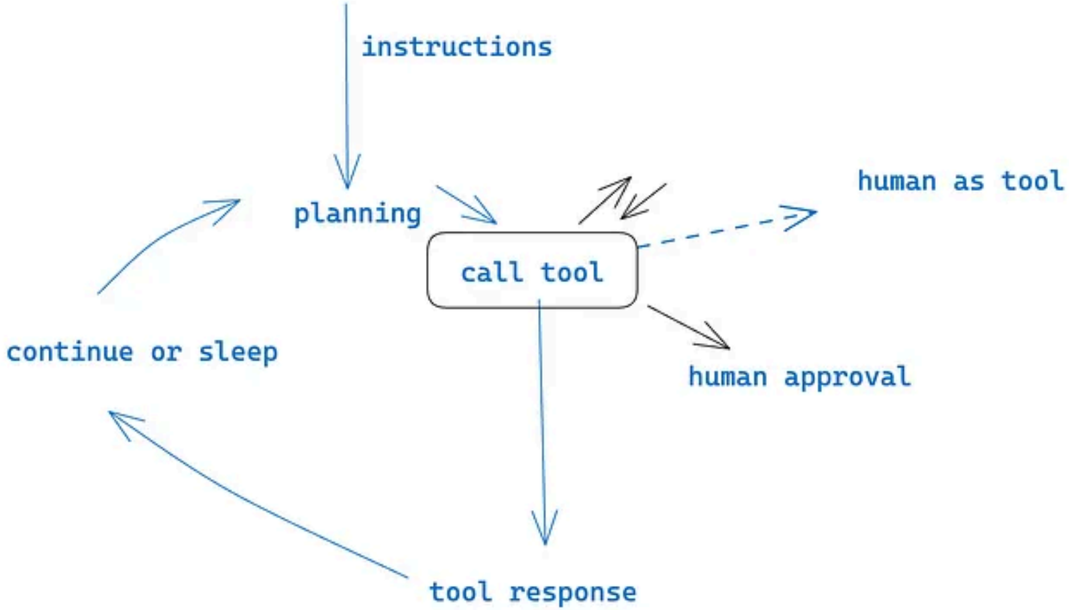
Planning and Reasoning and Routing



Tool Calling



Gen 3: Outer Loop



execute some set of instructions/tasks for minutes, hours, days, weeks or even longer

No fancy UI, no Proprietary Interface.

Meet users where they are

Slack, whatsapp, SMS(High urgency)

FACTOR 12: Agent systems are Stateless reducers

if you cannot recreate the exact current "state" and predict the "next action" of your agent by only looking at the historical sequence of events (your state), then you are violating Factor 12.

```python
class AgentState(TypedDict):
    events: Annotated[list[dict[str, Any]], operator.add]


def call_subagent(state: AgentState):
    # get the last sequence of type TASK
    last_task = None
    for event in reversed(state["events"]):
        if event.get("type") == "task":
            last_task = event
            break

    if last_task is None:
        return {
            "events": [{"type": "error", "message": "No task found to call subagent"}]
        }

    # Assume llm.invoke uses last_task to generate an answer
    answer = llm.invoke(last_task)
    return {"events": [{"type": "answer", "content": answer}]}


def check_answer(state: AgentState):
    # --- VIOLATION HERE ---
    retry_count = 0  # <--- This is local, ephemeral state
    # get the last answer from events
    last_answer = None
    for event in reversed(state["events"]):
        if event.get("type") == "answer":
            last_answer = event.get("content")
            break

    if some_condition != last_answer:
        retry_count += 1  # <--- Modifies local, ephemeral state
        # The agent *might* return something like:
        # return {"events": [{"type": "retry_request", "count": retry_count}]}
        # But 'retry_count' itself is not directly from 'state["events"]' for the *decision*
        return {
            "events": [{"type": "instruction", "action": "retry_task"}]
        }  # Example of directing graph

    # If the condition is met, perhaps return a completion event
    return {"events": [{"type": "status", "message": "Answer checked and correct"}]}
```

```python
class AgentState(TypedDict):
    events: Annotated[list[dict[str, Any]], operator.add]


# Assume call_subagent's return value is a state update, not a direct call
def call_subagent(state: AgentState):
    # get the last sequence of type TASK
    last_task = None
    for event in reversed(state["events"]):
        if event.get("type") == "task":
            last_task = event
            break

    if last_task is None:
        return {
            "events": [{"type": "error", "message": "No task found to call subagent"}]
        }

    # Assume llm.invoke uses last_task to generate an answer
    answer = llm.invoke(last_task)
    return {"events": [{"type": "answer", "content": answer}]}
```

```python
def check_answer(state: AgentState):
    current_retry_count = 0
    # Calculate retry_count by inspecting the logbook
    for event in state["events"]:
        if event.get("type") == "retry_attempt":
            current_retry_count += 1

    last_answer = None
    for event in reversed(state["events"]):
        if event.get("type") == "answer":
            last_answer = event.get("content")
            break

    # Assuming 'some_condition' is derived from the state or a fixed external value
    if some_condition != last_answer:
        # Decision is based on the inferred retry count
        if current_retry_count < MAX_RETRIES:  # MAX_RETRIES would be a constant
            # Add an event to the logbook to record the retry attempt
            return {
                "events": [
                    {"type": "retry_attempt", "count": current_retry_count + 1},
                    {
                        "type": "instruction",
                        "action": "retry_task",
                    },  # Direct the graph to retry
                ]
            }
        else:
            # Too many retries, escalate or mark as failed
            return {
                "events": [
                    {"type": "status", "message": "Max retries reached, task failed."},
                    {"type": "instruction", "action": "escalate_to_human"},
                ]
            }
    else:
        # Condition met, proceed
        return {"events": [{"type": "status", "message": "Answer checked and correct"
```

BUT WHY PUT SO MUCH EFFORT

Durability and Fault Tolerance:

- **Fault tolerance:** In real world scenarios, if your agent system crashes, gets deployed to new servers or gets restarted, it can continue from where it left off just by loading its previous state.
- **Violation Consequences:** If retry_count is a local variable, a crash means the agent forgets past retries. It might retry infinitely, or incorrectly assume it's the first attempt, leading to wasted resources or wrong decisions.

Reproducibility:

- **Time Travel Debugging:** If a user complains about strange agent behavior, you can load their exact thread_events log and re-run the agent's decision-making logic step-by-step.
- **Testing:** You can easily create test cases by providing specific thread_events sequences and assert the expected next action. No need to set up complex internal states.
- **Violation Consequences:** If retry_count is hidden, you can't accurately reproduce a bug unless you also know the exact value of retry_count at every point, which is usually impossible after a crash.

सी डैक
CDAC

Observability and Auditability:

- **Transparency:** For high-stakes agents (e.g., deploying to production, handling customer support), having this clear, auditable history is crucial for compliance, understanding "why" something happened, and building trust.
- **Violation Consequences:** If retry_count is a local variable, a crash means the agent forgets past retries. It might retry infinitely, or incorrectly assume it's the first attempt, leading to wasted resources or wrong decisions.

Concurrency and Simplicity (in a distributed context):

- **Easier Distributed Systems:** When an agent is spread across multiple services or even runs intermittently (e.g., processing an event, pausing, waiting for human, processing next event), a stateless reducer simplifies coordination. Each part of the system just appends to the logbook and reads the logbook to make decisions. There's no complex shared memory or locking for internal variables.
- **Reduced Mental Overhead:** The developer only needs to think about the sequence of events, not about how to keep multiple separate state variables synchronized.

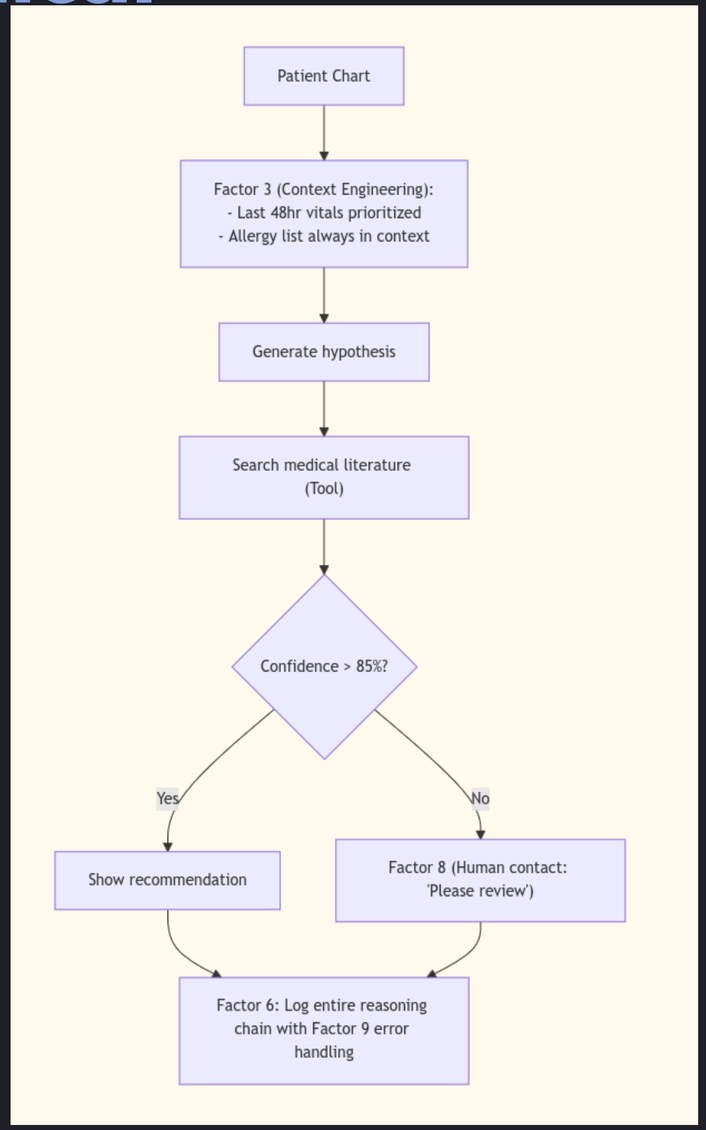
सी-डैक
CDAC

Anticipate and expect Non deterministic systems.

सी-डैक
CDAC

Example 1: Healthcare - Clinical Decision Support Agent

- Must cite medical literature for every recommendation
- Cannot act autonomously on critical decisions
- Handles sensitive data
- Works across shifts



Example 2: Finance - Fraud Investigation Agent

- A bank needs agents to investigate suspicious transactions, but:
- High stakes - false positives freeze customer accounts
- Regulatory requirements - must explain every decision to auditors
- Time-sensitive - fraud detection windows are narrow
- Complex tool chains - needs access to 8+ internal systems

Factor	How It Solves Finance Challenges
Factor 1 & 2: Tools as structured outputs	Ensures SQL queries to transaction DB are valid before execution; constrains account actions to approved types
Factor 8: Own your control flow	Implements "investigation protocol": check history → verify merchant → assess velocity → human review; never skips steps
Factor 9: Compact errors	When external merchant verification API fails: "Merchant API timeout (error_code: 503), proceeding with internal records only"
Factor 12: Stateless reducer	Each investigation is self-contained; enables parallel processing of 1000s of flagged transactions

“I feel like consistently, the most magical moments out of AI building come about for me when I'm really, really, really just close to the edge of the model capability.” -NotebookLM

So build something MAGICAL

सी डैक
CDAC