Agentic AI

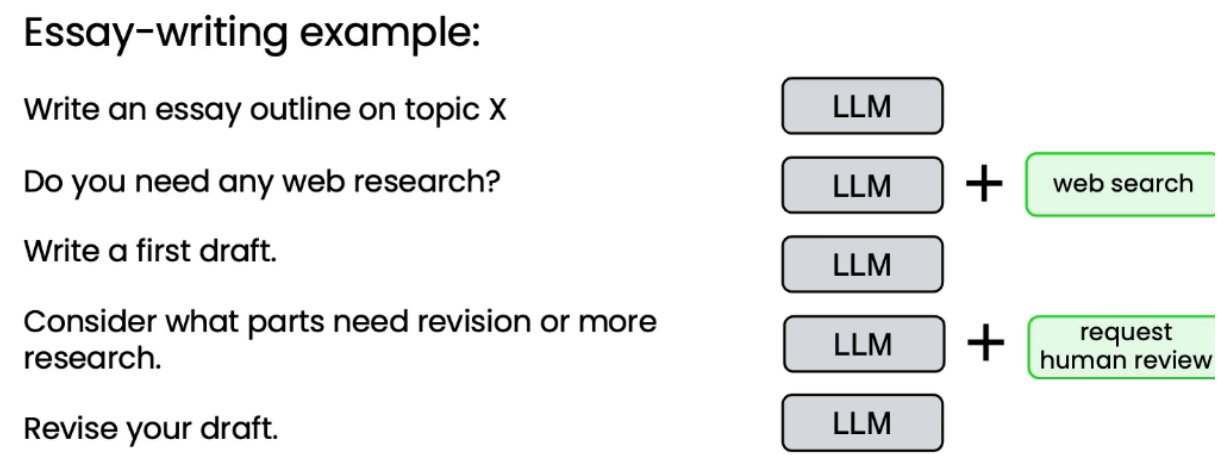
What is Agentic AI Workflow

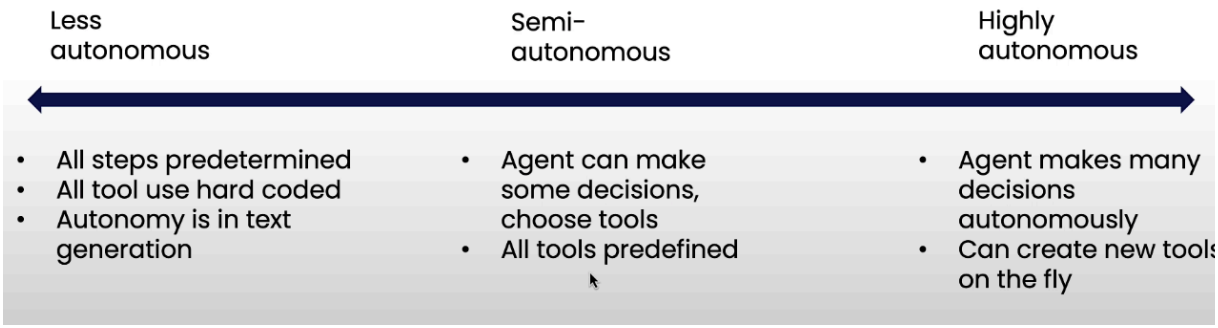
An agent is just a model with tools called in a loop.

Agents are just **"LLM + loop + tools"**

An agentic AI workflow is a process where an LLM-based app executes multiple steps to complete a task.

For example:



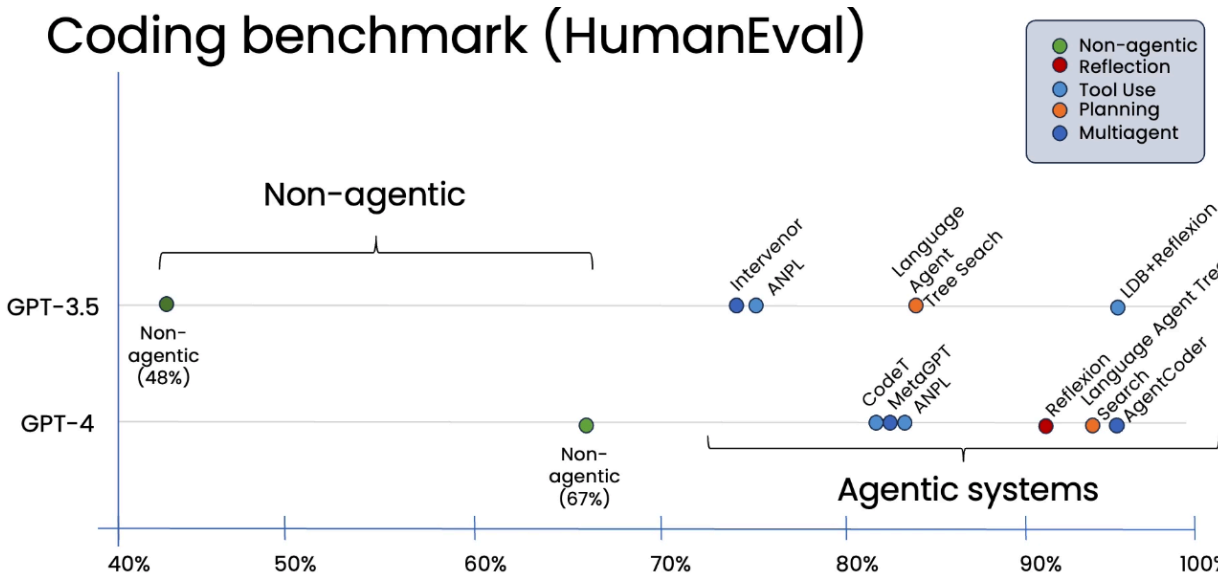


Key benefits of agentic workflows

1. Much better performance

Even a smaller model can have much improved performance with agentic workflow. Agentic workflows enable iterative improvement through reflection and revision, often delivering gains larger than upgrading to newer model

generations.



2. Faster than humans because of parallelization
3. Modular: can add or update tools, swap out models

Tasks agentic AI is suited to:

Clear, step-by-step process	Steps not known ahead of time
Standard procedures to follow	Plan/solve as you go
Text assets only	Multimodal (sound, vision)

Evaluation

Types of evaluations:

1. LLM-as-Judge for subjective evals.
2. Code-based evaluations for objective evals.
3. End-to-end output evals
4. Component wise evals.

Agent Design Patterns

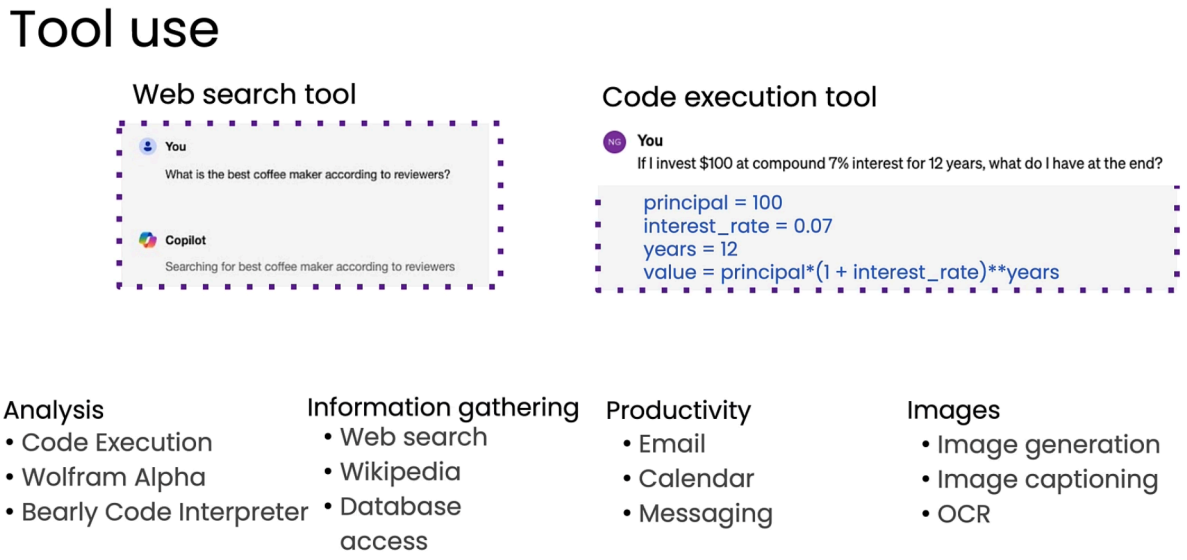
1. Reflection

Have LLM review its own output and revise it based on any issues it finds. It could be the same agent reviewing and revising the code or there could be a different critique agent that reviews the output of the code generating agent provides feedback based on which the code generating agent revises its output to address any issues.

The critic LLM could be a reasoning model.

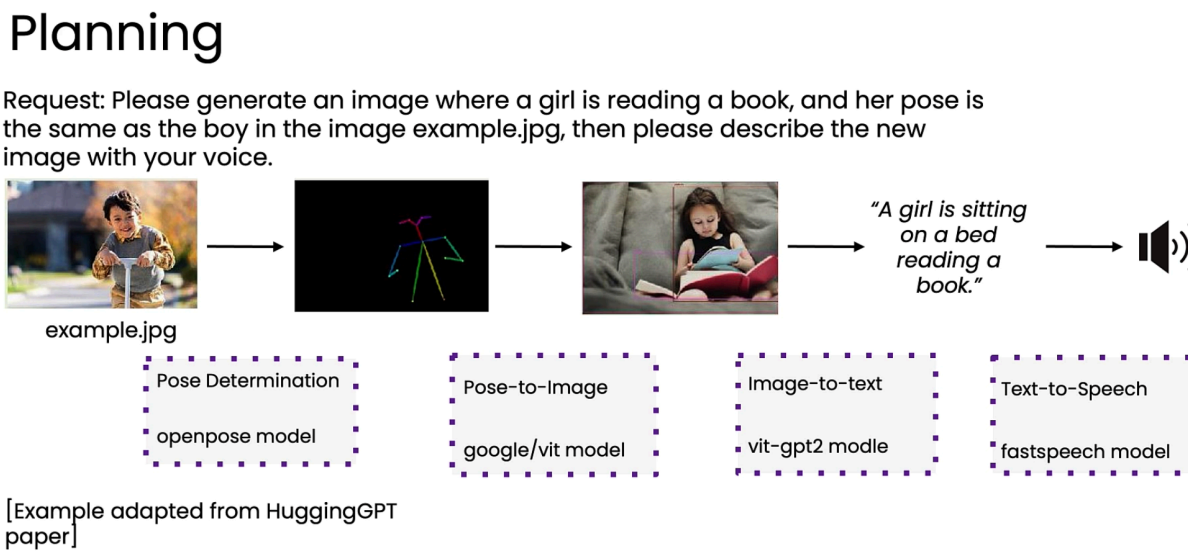
2. Tool Use

LLMs are given multiple tools to select for different tasks they have to perform.



3. Planning

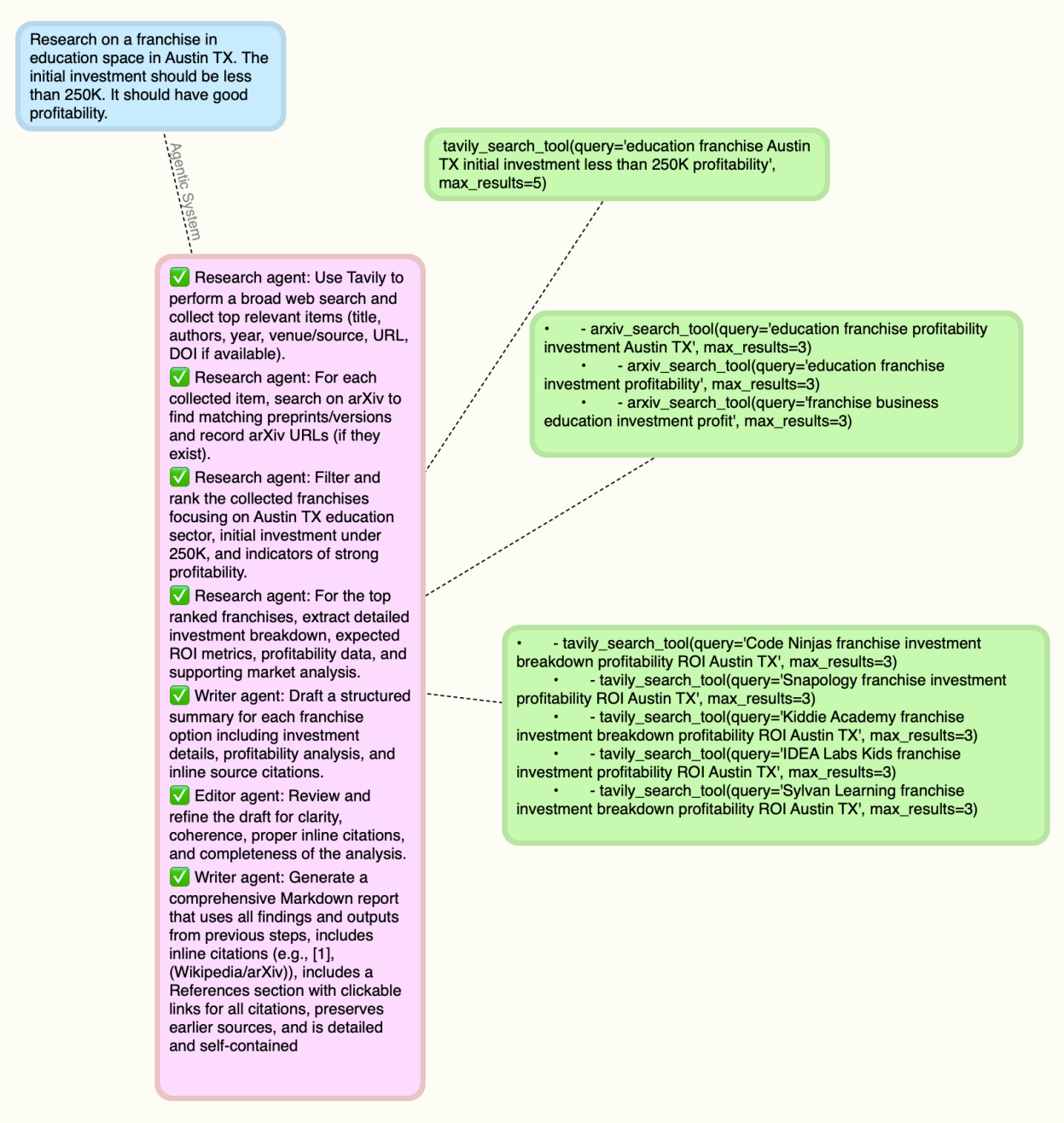
LLMs can use planning to solve a problem step-by-step and call different models in the process to achieve the end result.



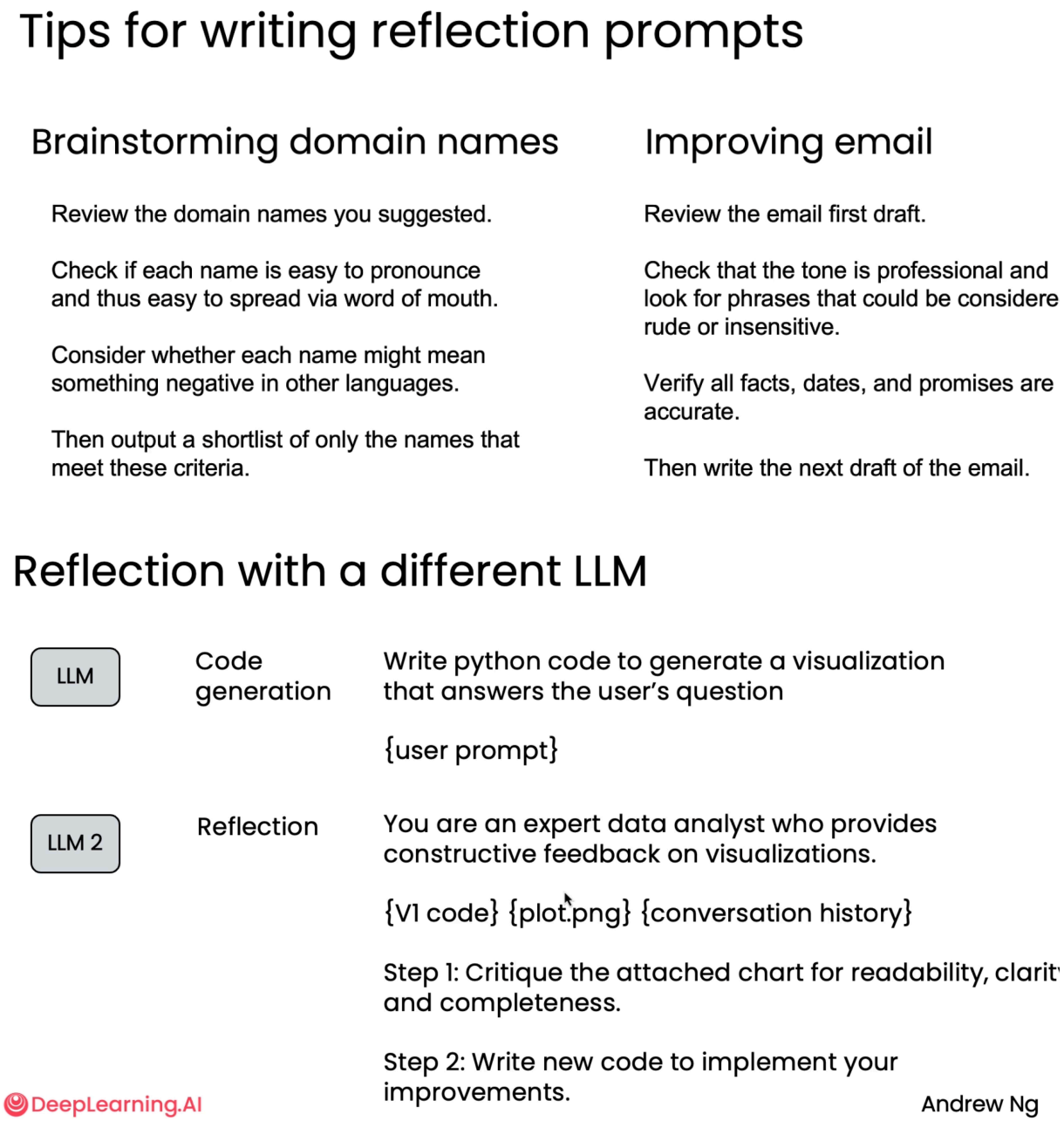
4. Multi-agent collaboration

Harder to control multi agent systems but they yield better performance than a single agent system.

Example of Research Agent



An LLM examines and critiques its own output, then generates improved versions, similar to how humans edit their work



Project 1 - Reflection Pattern - Agent to generate Chart

1. Prompt an LLM to write python code that generates a chart in response to a user query about the coffee dataset.
2. Coffee dataset includes fields such as date, coffee_type, quantity and revenue. The data frame schema is passed to LLM.
3. User query: Create a plot comparing Q1 coffee sales in 2024 and 2025 using the data in coffee_sales.csv.

```
def generate_chart_code(instruction: str, model: str,
out_path_v1: str) -> str:
    """Generate Python code to make a plot with matplotlib using
tag-based wrapping."""

    prompt = f"""
    You are a data visualization expert.

    Return your answer *strictly* in this format:

    <execute_python>
    # valid python code here
    </execute_python>

    Do not add explanations, only the tags and the code.

    The code should create a visualization from a DataFrame 'df'
with these columns:
    - date (M/D/YY)
    - time (HH:MM)
    - cash_type (card or cash)
    - card (string)
    - price (number)
    - coffee_name (string)
    - quarter (1-4)
    - month (1-12)
    - year (YYYY)

    User instruction: {instruction}

    Requirements for the code:
    1. Assume the DataFrame is already loaded as 'df'.
    2. Use matplotlib for plotting.
    3. Add clear title, axis labels, and legend if needed.
    4. Save the figure as '{out_path_v1}' with dpi=300.
    5. Do not call plt.show().
    6. Close all plots with plt.close().
    7. Add all necessary import python statements

    Return ONLY the code wrapped in <execute_python> tags.
    """

    response = utils.get_response(model, prompt)
    return response

# Generate initial code
code_v1 = generate_chart_code(
    instruction="Create a plot comparing Q1 coffee sales in 2024
```

```
and 2025 using the data in coffee_sales.csv.",
    model="gpt-4o-mini",
    out_path_v1="chart_v1.png"
)

utils.print_html(code_v1, title="LLM output with first draft
code")
```

The LLM outputs:

```
<execute_python>
import pandas as pd
import matplotlib.pyplot as plt

# Filter the DataFrame for Q1 sales in 2024 and 2025
df['date'] = pd.to_datetime(df['date'])
q1_sales = df[(df['quarter'] == 1) & (df['year'].isin([2024,
2025]))]

# Group by year and coffee_name and sum prices
sales_summary = q1_sales.groupby(['year', 'coffee_name'])
['price'].sum().unstack()

# Plotting
plt.figure(figsize=(10, 6))
sales_summary.plot(kind='bar', figsize=(10, 6))
plt.title('Q1 Coffee Sales Comparison: 2024 vs 2025')
plt.xlabel('Coffee Name')
plt.ylabel('Total Sales ($)')
plt.legend(title='Year')
plt.tight_layout()

# Save the figure
plt.savefig('chart_v1.png', dpi=300)
plt.close()
</execute_python>
```

1. Reflect on the output
    - Provide chart to a multi-modal LLM
    - LLM will analyze the chart visually - The LLM visually examines the generated chart to identify flaws and suggest better visualization types
    - LLM suggests improvements - for example, fixing axis labels, adjusting the chart type, improving color choices, or highlighting missing legends.

```
def reflect_on_image_and_regenerate(
    chart_path: str,
    instruction: str,
    model_name: str,
    out_path_v2: str,
    code_v1: str,
) -> tuple[str, str]:
    """
    Critique the chart IMAGE and the original code against the
instruction,
    then return refined matplotlib code.
    Returns (feedback, refined_code_with_tags).
    Supports OpenAI and Anthropic (Claude).
    """
    media_type, b64 = utils.encode_image_b64(chart_path)


    prompt = f"""
    You are a data visualization expert.
    Your task: critique the attached chart and the original code
against the given instruction,
    then return improved matplotlib code.

    Original code (for context):
    {code_v1}

    OUTPUT FORMAT (STRICT):
    1) First line: a valid JSON object with ONLY the "feedback"
field.
    Example: {{"feedback": "The legend is unclear and the axis
labels overlap."}}

    2) After a newline, output ONLY the refined Python code
wrapped in:
    <execute_python>
    ...
    </execute_python>

    3) Import all necessary libraries in the code. Don't assume
any imports from the original code.

    HARD CONSTRAINTS:
    - Do NOT include Markdown, backticks, or any extra prose
outside the two parts above.
    - Use pandas/matplotlib only (no seaborn).
    - Assume df already exists; do not read from files.
    - Save to '{out_path_v2}' with dpi=300.
    - Always call plt.close() at the end (no plt.show()).
    - Include all necessary import statements.
```

```
    Schema (columns available in df):
    - date (M/D/YY)
    - time (HH:MM)
    - cash_type (card or cash)
    - card (string)
    - price (number)
    - coffee_name (string)
    - quarter (1-4)
    - month (1-12)
    - year (YYYY)

    Instruction:
    {instruction}
    """


    # In case the name is "Claude" or "Anthropic", use the safe
helper
    lower = model_name.lower()
    if "claude" in lower or "anthropic" in lower:
        # ✅ Use the safe helper that joins all text blocks and
adds a system prompt
        content = utils.image_anthropic_call(model_name, prompt,
media_type, b64)
    else:
        content = utils.image_openai_call(model_name, prompt,
media_type, b64)

    # --- Parse ONLY the first JSON line (feedback) ---
    lines = content.strip().splitlines()
    json_line = lines[0].strip() if lines else ""

    try:
        obj = json.loads(json_line)
    except Exception as e:
        # Fallback: try to capture the first {...} in all the
content
        m_json = re.search(r"\{.*?\}", content, flags=re.DOTALL)
        if m_json:
            try:
                obj = json.loads(m_json.group(0))
            except Exception as e2:
                obj = {"feedback": f"Failed to parse JSON: {e2}",
"refined_code": ""}
        else:
            obj = {"feedback": f"Failed to find JSON: {e}",
"refined_code": ""}
```

```
    # --- Extract refined code from <execute_python>...</
execute_python> ---
    m_code = re.search(r"<execute_python>([\s\S]*?)</
execute_python>", content)
    refined_code_body = m_code.group(1).strip() if m_code else ""
    refined_code =
utils.ensure_execute_python_tags(refined_code_body)

    feedback = str(obj.get("feedback", "")).strip()
    return feedback, refined_code
```

1. Execute the python code generated by critic LLM.
2. Putting it all together - creating an end-to-end workflow

```
def run_workflow(
    dataset_path: str,
    user_instructions: str,
    generation_model: str,
    reflection_model: str,
    image_basename: str = "chart",
):
    """
    End-to-end pipeline:
      1) load dataset
      2) generate V1 code
      3) execute V1 → produce chart_v1.png
      4) reflect on V1 (image + original code) → feedback +
refined code
      5) execute V2 → produce chart_v2.png

    Returns a dict with all artifacts (codes, feedback, image
paths).
    """
    # 0) Load dataset; utils handles parsing and feature
derivations (e.g., year/quarter)
    df = utils.load_and_prepare_data(dataset_path)
    utils.print_html(df.sample(n=5), title="Random Sample of
Dataset")

    # Paths to store charts
    out_v1 = f"{image_basename}_v1.png"
    out_v2 = f"{image_basename}_v2.png"

    # 1) Generate code (V1)
    utils.print_html("Step 1: Generating chart code (V1)… 📈")
    code_v1 = generate_chart_code(
        instruction=user_instructions,
```

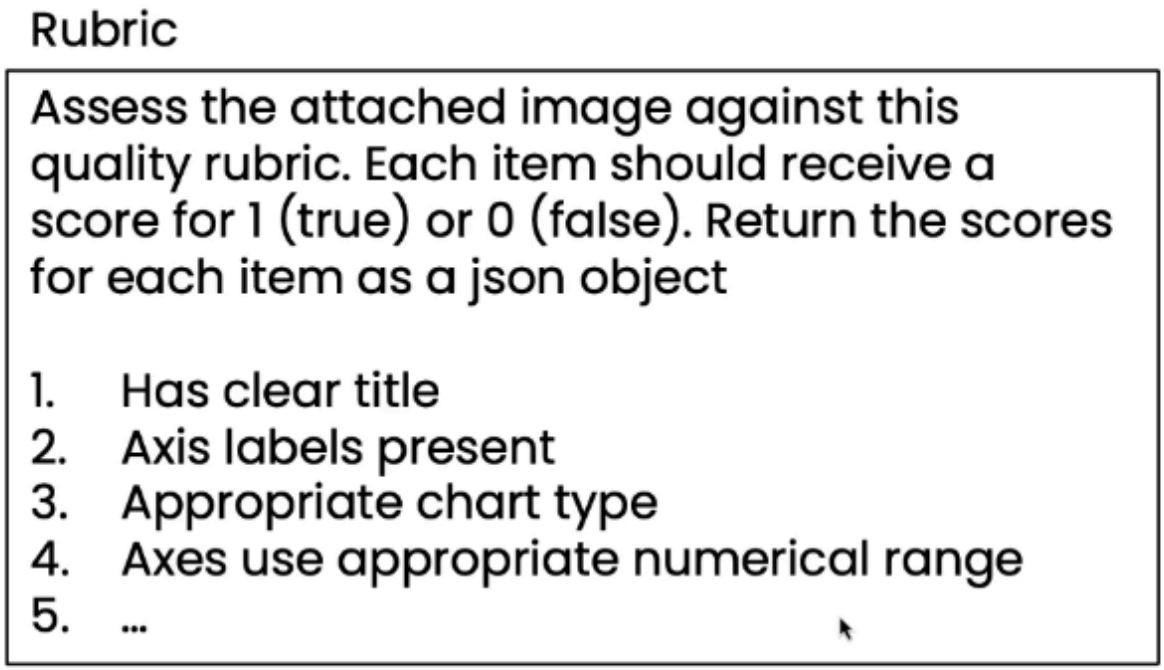
```
        model=generation_model,
        out_path_v1=out_v1,
    )
    utils.print_html(code_v1, title="LLM output with first draft
code (V1)")

    # 2) Execute V1 (hard-coded: extract <execute_python> block
and run immediately)
    utils.print_html("Step 2: Executing chart code (V1)… 💻")
    match = re.search(r"<execute_python>([\s\S]*?)</
execute_python>", code_v1)
    if match:
        initial_code = match.group(1).strip()
        exec_globals = {"df": df}
        exec(initial_code, exec_globals)
    utils.print_html(out_v1, is_image=True, title="Generated
Chart (V1)")

    # 3) Reflect on V1 (image + original code) to get feedback
and refined code (V2)
    utils.print_html("Step 3: Reflecting on V1 (image + code) and
generating improvements… 🔁")
    feedback, code_v2 = reflect_on_image_and_regenerate(
        chart_path=out_v1,
        instruction=user_instructions,
        model_name=reflection_model,
        out_path_v2=out_v2,
        code_v1=code_v1,  # pass original code for context
    )
    utils.print_html(feedback, title="Reflection feedback on V1")
    utils.print_html(code_v2, title="LLM output with revised code
(V2)")

    # 4) Execute V2 (hard-coded: extract <execute_python> block
and run immediately)
    utils.print_html("Step 4: Executing refined chart code (V2)…
🖼️")
    match = re.search(r"<execute_python>([\s\S]*?)</
execute_python>", code_v2)
    if match:
        reflected_code = match.group(1).strip()
        exec_globals = {"df": df}
        exec(reflected_code, exec_globals)
    utils.print_html(out_v2, is_image=True, title="Regenerated
Chart (V2)")

    return {
        "code_v1": code_v1,
```

```
        "chart_v1": out_v1,
        "feedback": feedback,
        "code_v2": code_v2,
        "chart_v2": out_v2,
    }

# Here, insert your updates
user_instructions="Create a plot comparing Q1 coffee sales in
2024 and 2025 using the data in coffee_sales.csv." # write your
instruction here
generation_model="gpt-4o-mini"
reflection_model="o4-mini"
image_basename="drink_sales"

# Run the complete agentic workflow
_ = run_workflow(
    dataset_path="coffee_sales.csv",
    user_instructions=user_instructions,
    generation_model=generation_model,
    reflection_model=reflection_model,
    image_basename=image_basename
)
```

Evaluating the impact of Reflection workflows

1. Grading with a rubric gives more consistent results

Use a rubric with specific binary criteria to evaluate individual outputs, then sum the scores



Within the rubric to get more consistent result, have each criteria be evaluated as binary true or false. Then add up the numbers to get a score between 1-10 for eg.

2. Have an evaluation dataset with questions and expected answers.

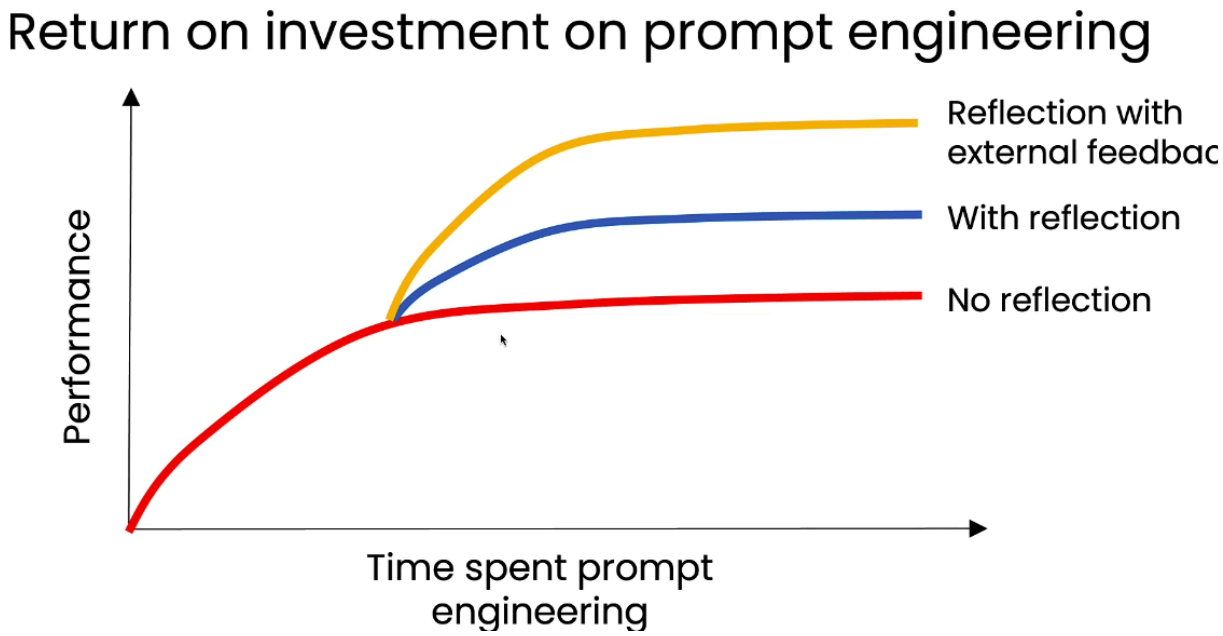
3. Run each time you change the prompts to LLM.

4. Code-based evals are easier to manage - works best for objective evals.

5. LLM-as-a-judge evals are used for subjective evals like say evaluating a chart or text summary. Rubric based grading is better as explained above.

Reflection with External Feedback

LLM using some external source of information to reflect will yield even higher performance gain.



Other examples of tools to help reflection

Challenge	Example	Source of feedback
Mentioning competitors	Our company's shoes are better than RivalCo	Pattern matching for competitor names
Fact checking an essay	The Taj Mahal was built in 1648	Web search results
LLM won't follow output length guidelines	Essay is over word limit	Word count tool

Project 2 - Reflection Pattern - Agent to generate SQL

Using reflection to improve an agentic workflow that converts questions to database SQL queries.

```
def generate_sql(question: str, schema: str, model: str) -> str:
    prompt = f"""
    You are a SQL assistant. Given the schema and the user's
question, write a SQL query for SQLite.
```

```
        Schema:
        {schema}

        User question:
        {question}

        Respond with the SQL only.
        """
        response = client.chat.completions.create(
            model=model,
            messages=[{"role": "user", "content": prompt}],
            temperature=0,
        )
        return response.choices[0].message.content.strip()

    # Example usage of generate_sql

    # We provide the schema as a string
    schema = """
    Table name: transactions
    id (INTEGER)
    product_id (INTEGER)
    product_name (TEXT)
    brand (TEXT)
    category (TEXT)
    color (TEXT)
    action (TEXT)
    qty_delta (INTEGER)
    unit_price (REAL)
    notes (TEXT)
    ts (DATETIME)
    """

    # We ask a question about the data in natural language
    question = "Which color of product has the highest total sales?"

    utils.print_html(question, title="User Question")

    # Generate the SQL query using the specified model
    sql_V1 = generate_sql(question, schema, model="openai:gpt-4.1")

    # Display the generated SQL query
    utils.print_html(sql_V1, title="SQL Query V1")
```

Generated SQL Query (V1)

```sql
SELECT color, SUM(qty_delta * unit_price) AS total_sales
```

```
FROM transactions
WHERE action = 'sale'
GROUP BY color
ORDER BY total_sales DESC
LIMIT 1;
```
```

Reflect on the query:

```
    def refine_sql(
        question: str,
        sql_query: str,
        schema: str,
        model: str,
) -> tuple[str, str]:
        """
        Reflect on whether a query's *shown output* answers the
    question,
        and propose an improved SQL if needed.
        Returns (feedback, refined_sql).
        """
        prompt = f"""
    You are a SQL reviewer and refiner.

    User asked:
    {question}

    Original SQL:
    {sql_query}

    Table Schema:
    {schema}

    Step 1: Briefly evaluate if the SQL OUTPUT fully answers the
    user's question.
    Step 2: If improvement is needed, provide a refined SQL query for
    SQLite.
    If the original SQL is already correct, return it unchanged.

    Return STRICT JSON with two fields:
    {{
      "feedback": "<1-3 sentences explaining the gap or confirming
    correctness>",
      "refined_sql": "<final SQL to run>"
    }}
    """
```
```

```
        response = client.chat.completions.create(
            model=model,
            messages=[{"role": "user", "content": prompt}],
            temperature=0,
        )

        content = response.choices[0].message.content
        try:
            obj = json.loads(content)
            feedback = str(obj.get("feedback", "")).strip()
            refined_sql = str(obj.get("refined_sql",
    sql_query)).strip()
            if not refined_sql:
                refined_sql = sql_query
        except Exception:
            # Fallback if model doesn't return valid JSON
            feedback = content.strip()
            refined_sql = sql_query

        return feedback, refined_sql
```

Feedback on V1

```
The SQL correctly attempts to find the color with the
highest total sales, but negative total_sales suggests
qty_delta might be negative for sales. Typically, 'sale'
transactions have qty_delta < 0 (e.g., -1 per item sold),
so multiplying qty_delta * unit_price gives a negative
value. To calculate total sales in monetary terms, use
ABS(qty_delta) or multiply by -1. The refined SQL fixes
this.
```

Refine the query based on reflection feedback:

```
    def refine_sql_external_feedback(
        question: str,
        sql_query: str,
        df_feedback: pd.DataFrame,
        schema: str,
        model: str,
    ) -> tuple[str, str]:
        """
        Evaluate whether the SQL result answers the user's question
    and,
        if necessary, propose a refined version of the query.
        Returns (feedback, refined_sql).
        """
```

```
    prompt = f"""
    You are a SQL reviewer and refiner.

    User asked:
    {question}

    Original SQL:
    {sql_query}

    SQL Output:
    {df_feedback.to_markdown(index=False)}

    Table Schema:
    {schema}

    Step 1: Briefly evaluate if the SQL output answers the user's
question.
    Step 2: If the SQL could be improved, provide a refined SQL
query.
    If the original SQL is already correct, return it unchanged.

    Return a strict JSON object with two fields:
    - "feedback": brief evaluation and suggestions
    - "refined_sql": the final SQL to run
    """

    response = client.chat.completions.create(
        model=model,
        messages=[{"role": "user", "content": prompt}],
        temperature=1.0,
    )


    content = response.choices[0].message.content
    try:
        obj = json.loads(content)
        feedback = str(obj.get("feedback", "")).strip()
        refined_sql = str(obj.get("refined_sql",
sql_query)).strip()
        if not refined_sql:
            refined_sql = sql_query
    except Exception:
        # Fallback if the model does not return valid JSON:
        # use the raw content as feedback and keep the original
SQL
        feedback = content.strip()
        refined_sql = sql_query

    return feedback, refined_sql
```

Refined SQL Query (V2)

```
SELECT color, SUM(ABS(qty_delta) * unit_price) AS
total_sales
FROM transactions
WHERE action = 'sale'
GROUP BY color
ORDER BY total_sales DESC
LIMIT 1;

Database Query Workflow

def run_sql_workflow(
    db_path: str,
    question: str,
    model_generation: str = "openai:gpt-4.1",
    model_evaluation: str = "openai:gpt-4.1",
):
    """
    End-to-end workflow to generate, execute, evaluate, and
refine SQL queries.

    Steps:
      1) Extract database schema
      2) Generate SQL (V1)
      3) Execute V1 → show output
      4) Reflect on V1 with execution feedback → propose
refined SQL (V2)
      5) Execute V2 → show final answer
    """

    # 1) Schema
    schema = utils.get_schema(db_path)
    utils.print_html(
        schema,
        title="📘 Step 1 — Extract Database Schema"
    )

    # 2) Generate SQL (V1)
    sql_v1 = generate_sql(question, schema,
model_generation)
```

```
    utils.print_html(
        sql_v1,
        title="🧠 Step 2 — Generate SQL (V1)"
    )

    # 3) Execute V1
    df_v1 = utils.execute_sql(sql_v1, db_path)
    utils.print_html(
        df_v1,
        title="🧪 Step 3 — Execute V1 (SQL Output)"
    )

    # 4) Reflect on V1 with execution feedback → refine to
V2
    feedback, sql_v2 = refine_sql_external_feedback(
        question=question,
        sql_query=sql_v1,
        df_feedback=df_v1,          # external feedback:
real output of V1
        schema=schema,
        model=model_evaluation,
    )
    utils.print_html(
        feedback,
        title="🧭 Step 4 — Reflect on V1 (Feedback)"
    )
    utils.print_html(
        sql_v2,
        title="🔁 Step 4 — Refined SQL (V2)"
    )

    # 5) Execute V2
    df_v2 = utils.execute_sql(sql_v2, db_path)
    utils.print_html(
        df_v2,
        title="✅ Step 5 — Execute V2 (Final Answer)"
    )
```

```
run_sql_workflow(
    "products.db",
    "Which color of product has the highest total sales?",
    model_generation="openai:gpt-4.1",
    model_evaluation="openai:gpt-4.1"
)
```

Project 3- Reflection Pattern - Write an essay on a topic

```
from dotenv import load_dotenv

load_dotenv()

import aisuite as ai

# Define the client. You can use this variable inside your
graded functions!
CLIENT = ai.Client()
def generate_draft(topic: str, model: str =
"openai:gpt-4o") -> str:

    ### START CODE HERE ###

    # Define your prompt here. A multi-line f-string is
typically used for this.
    prompt = f"""You are an academic essay writer. Based on
the following simple prompt, generate a complete first
draft of an essay. The essay should include a clear
introduction with a thesis statement, 2–3 well-developed
body paragraphs with topic sentences and supporting points,
and a concise conclusion that summarizes the main argument.
Write in a formal yet accessible tone, and aim for clarity
and logical flow.

Simple prompt: {topic}

Before writing, restate the topic in your own words, and
then produce the essay in standard academic formatting (no
bullet points). Keep it between 600–900 words.
```

```
    """

    ### END CODE HERE ###

    # Get a response from the LLM by creating a chat with
the client.
    response = CLIENT.chat.completions.create(
        model=model,
        messages=[{"role": "user", "content": prompt}],
        temperature=1.0,
    )

    return response.choices[0].message.content

def reflect_on_draft(draft: str, model: str = "openai:o4-
mini") -> str:

    ### START CODE HERE ###

    # Define your prompt here. A multi-line f-string is
typically used for this.
    prompt = f"""
    Reflect critically on the following essay draft you
just generated. Evaluate it as if you were a writing
instructor. Identify the essay's main strengths (clarity,
coherence, argument structure, evidence, style) and its
weaknesses (gaps in reasoning, lack of depth, vague
language, structural issues).. Suggest at least three
specific ways to improve the essay's overall quality—such
as enhancing the thesis, refining transitions, or deepening
analysis. Then, summarize in one paragraph how these
improvements would make the essay more persuasive and
polished.
    Draft: {draft}

    """

    ### END CODE HERE ###

    # Get a response from the LLM by creating a chat with
```

```
the client.
    response = CLIENT.chat.completions.create(
        model=model,
        messages=[{"role": "user", "content": prompt}],
        temperature=1.0,
    )

    return response.choices[0].message.content

def revise_draft(original_draft: str, reflection: str,
model: str = "openai:gpt-4o") -> str:

    ### START CODE HERE ###

    # Define your prompt here. A multi-line f-string is
typically used for this.
    prompt = f"""Using the following essay draft and the
accompanying reflection and feedback, produce a revised
version of the essay.

Carefully address each weakness and suggestion for
improvement noted in the reflection.

Strengthen the thesis statement, improve coherence and
structure, add depth and specificity to arguments, and
enhance clarity and style as recommended.

Incorporate any additional points from the feedback into
the body and conclusion as appropriate.

Ensure the revised essay is well-organized, logically flows
from introduction to conclusion, and is polished and
persuasive.

Essay draft:
{original_draft}

Reflection and feedback:
{reflection}
```

```
Write the improved essay in standard academic format (no
bullet points).

    """

    # Get a response from the LLM by creating a chat with
the client.
    response = CLIENT.chat.completions.create(
        model=model,
        messages=[{"role": "user", "content": prompt}],
        temperature=1.0,
    )

    ### END CODE HERE ###

    return response.choices[0].message.content
```

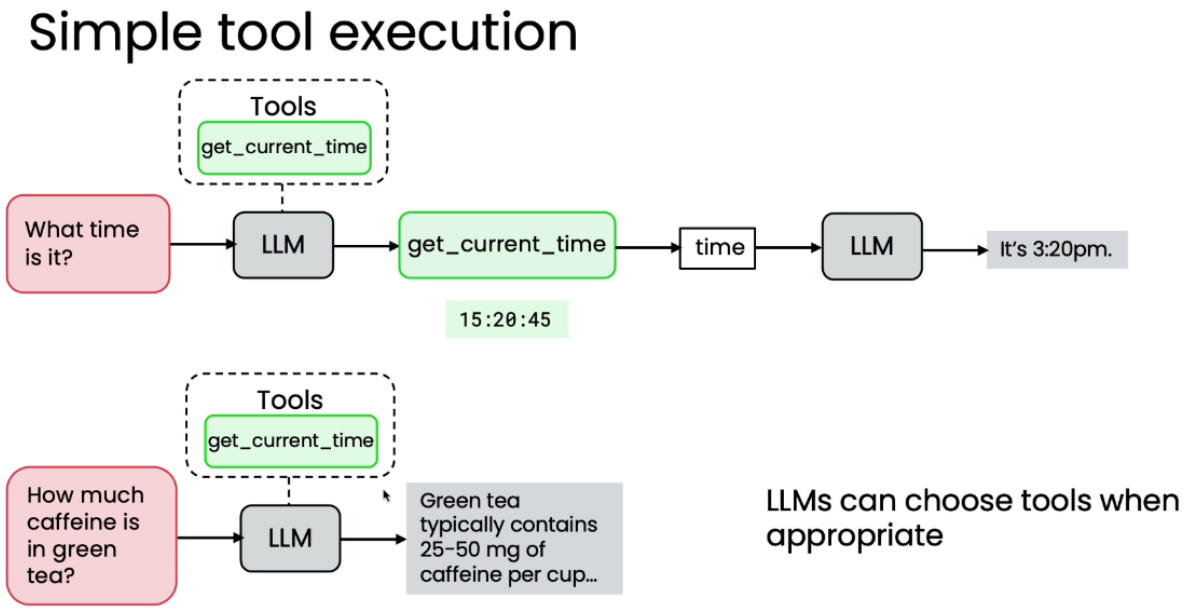
Execute the essay writing agentic workflow

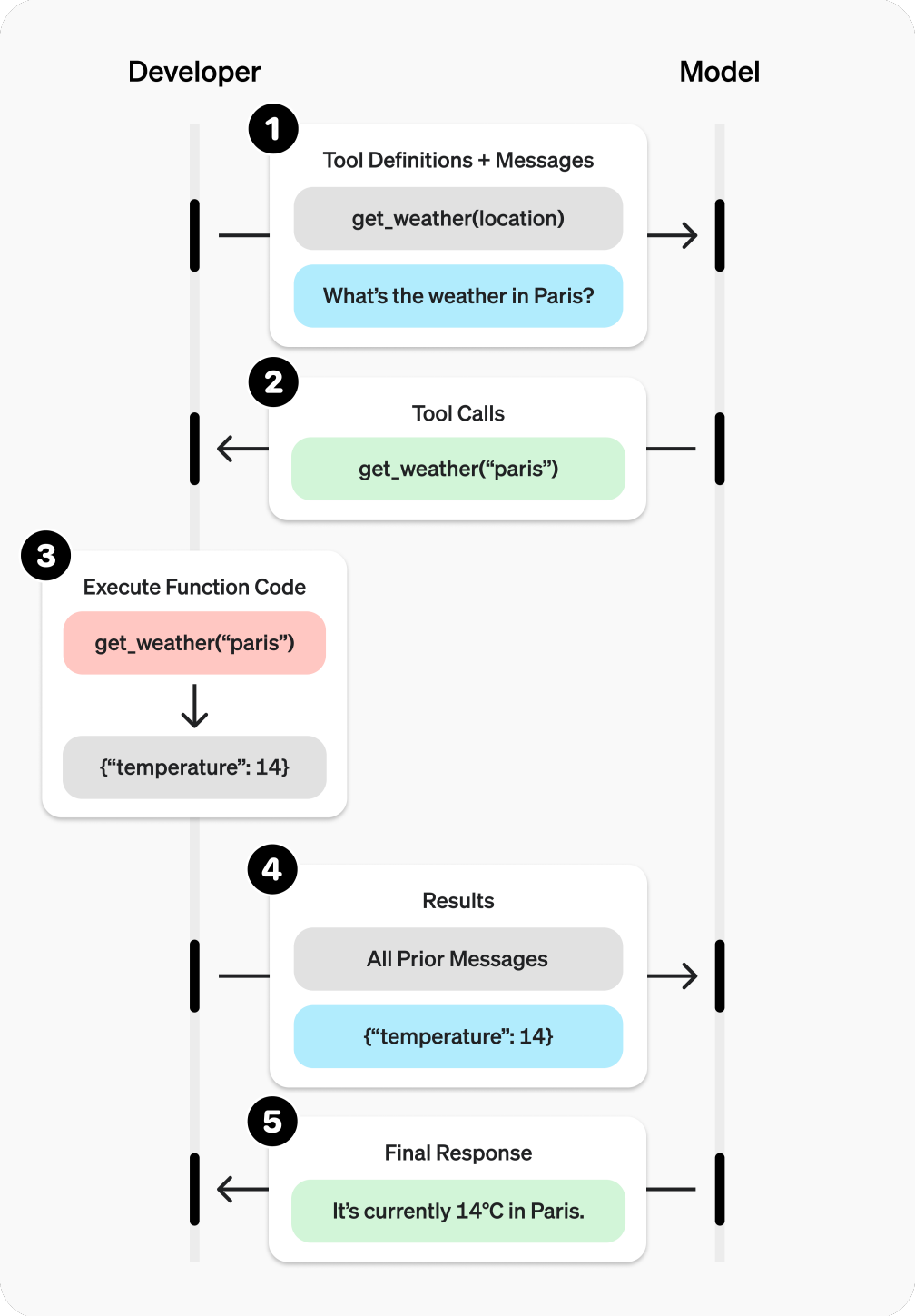
```
essay_prompt = "Should social media platforms be regulated
by the government?"

# Agent 1 — Draft
draft = generate_draft(essay_prompt)
print("📝 Draft:\n")
print(draft)

# Agent 2 — Reflection
feedback = reflect_on_draft(draft)
print("\n🧠 Feedback:\n")
print(feedback)

# Agent 3 — Revision
revised = revise_draft(draft, feedback)
print("\n✍️ Revised:\n")
print(revised)
```

Tool Use

Think of tools as the agent's **actuators**: you give a natural language instruction ("check unread emails from my boss and send a polite reply"), and the model chooses **which tools** to call and **in what order** to complete the task.

- **Tool calling** allows LLMs to go beyond text generation—enabling them to call functions (tools) and complete multi-step tasks.
- The set of available tools determines what the agent can and cannot do (e.g., without `delete_email`, it cannot remove messages).
- Clear docstrings and consistent behavior help the LLM select the right tool for each step.
- AISuite manages the interaction layer: exposing Python functions as tools, accepting parameters, making API requests, and returning results.
- Observing the full workflow—from prompt → tool calls → outputs → final response—is key to understanding and improving how agents reason and act.





The tool calling flow

Tool calling is a multi-step conversation between your application and a model via the OpenAI API. The tool calling flow has five high level steps:

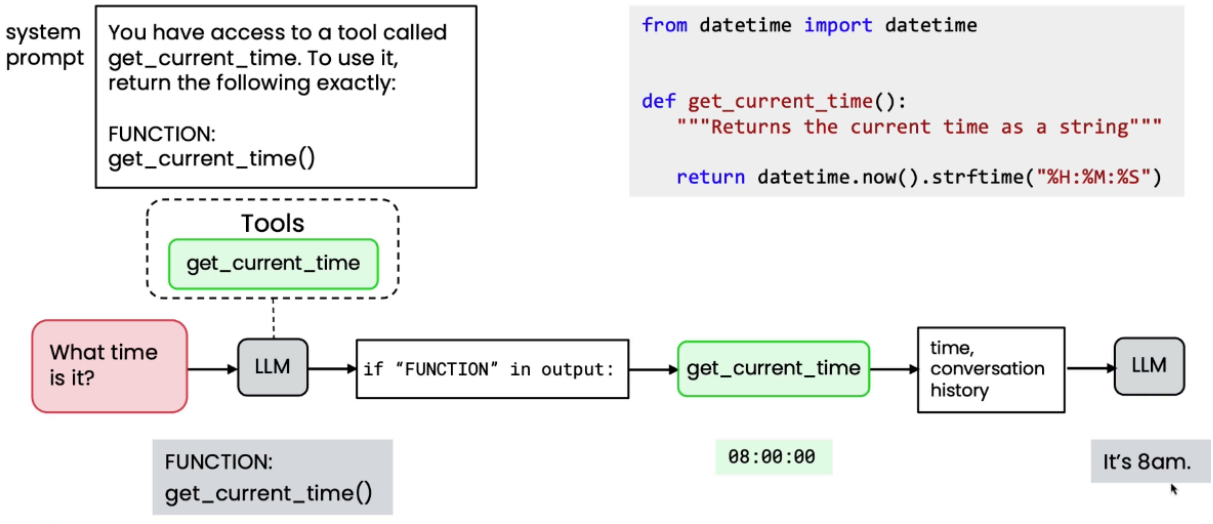
1. Make a request to the model with tools it could call
2. Receive a tool call from the model
3. Execute code on the application side with input from the tool call
4. Make a second request to the model with the tool output
5. Receive a final response from the model (or more tool calls)

Examples

Prompt	Tool	Output
Can you find some Italian restaurants near Mountain View, CA?	`web_search(query="restaurants near Mountain View, CA")`	Spaghetti City is an Italian restaurant in Mountain View…
Show me customers who bought white sunglasses	`query_database(table="sales", product="sunglasses", color="white")`	28 customers bought white sunglasses. Here they are…

How much money will I have after 10 years if I deposit $500 at 5% interest?	`interest_calc(principal=500, interest_rate=5, years=10)`
	OR
	`eval("500 * (1 + 0.05) ** 10")`
		$814.45

Prompting an LLM to use tools



The above prompting is no more needed as LLMs today are trained to call

tools. Earlier we needed to tell the LLM in its system prompt that a tool was available to them.

Note: LLM does not call the tool, it selects the right tool to be called and the params it needs to be passed and eventually the agent client will call the tool.

The modern day LLMs can just read the doc string of the function to be called and with that it can come to know when it should be calling the function.

Defining tools syntax

```python
from datetime import datetime


def get_current_time():

    """Returns the current time as a string"""

    return datetime.now().strftime("%H:%M:%S")
```

```python
import aisuite as ai
client = ai.Client()

response = client.chat.completions.create(
    model="openai:gpt-4o",
    messages=messages,
    tools=[get_current_time],
    max_turns=5
)
```

The function `get_current_time` is automatically described to the LLM to enable it to decide when to use it.

Behind the scenes (functions with parameters)

```python
from datetime import datetime
from zoneinfo import ZoneInfo


def get_current_time(timezone):
    """Returns current time for the given time zone"""
    timezone = ZoneInfo(timezone)
    return datetime.now(timezone).strftime("%H:%M:%S")
```

```python
import aisuite as ai
client = ai.Client()

response = client.chat.completions.create(
    model="openai:gpt-4o",
    messages=messages,
    tools=[get_current_time],
    max_turns=5
)
```

JSON Schema

```
tools = [{ "type": "function",

          "function": {"name" : "get_current_time",

                       "description": "Returns current time
                        for the given timezone.

                       "parameters": {
                                    "timezone": {
                                    "type": "string",
                                    "description": "The IANA
                                        time zone string, e.g
                                        'America/New_York' or
                                        'Pacific/Auckland'."
                                     }
                        }
          }
}]
```

Note: aisuite package only takes the name of the function and behind the scene it creates a JSON schema with tool name, description and params description. This JSON schema is passed to LLM.

Alternatively one can pass the JSON schema explicitly as well.

```
def will_it_rain(location: str, time_of_day: str):
    """Check if it will rain in a location at a given
time today.

    Args:
        location (str): Name of the city
        time_of_day (str): Time of the day in HH:MM
format.
    """
    return "YES"

tools = [{
    "type": "function",
    "function": {
        "name": "will_it_rain",
        "description": "Check if it will rain in a
location at a given time today",
        "parameters": {
            "type": "object",
            "properties": {
                "location": {
                    "type": "string",
                    "description": "Name of the city"
                },
                "time_of_day": {
                    "type": "string",
                    "description": "Time of the day in
HH:MM format."
                }
            },
            "required": ["location", "time_of_day"]
        }
    }
}]

response = client.chat.completions.create(
    model="openai:gpt-4o",
    messages=messages,
```

```
    tools=tools
)
```

Project 1 - Turning functions into tools

```
import json
import display_functions
from dotenv import load_dotenv
_ = load_dotenv()
import aisuite as ai

# Create an instance of the AISuite client
client = ai.Client()

from datetime import datetime

def get_current_time():
    """
    Returns the current time as a string.
    """
    return datetime.now().strftime("%H:%M:%S")

# Message structure
prompt = "What time is it?"
messages = [
    {
        "role": "user",
        "content": prompt,
    }
]
response = client.chat.completions.create(
    model="openai:gpt-4o",
    messages=messages,
    tools=[get_current_time],
    max_turns=5
)

# See the LLM response
print(response.choices[0].message.content)
```

Manually defining tools

```
tools = [{
    "type": "function",
    "function": {
        "name": "get_current_time", # <--- Your functions name
```

```
        "description": "Returns the current time as a string.", #
<--- a description for the LLM
        "parameters": {}
    }
}]
response = client.chat.completions.create(
    model="openai:gpt-4o",
    messages=messages,
    tools=tools, # <-- Your list of tools with get_current_time
    # max_turns=5 # <-- When defining tools manually, you must
handle calls yourself and cannot use max_turns
)
print(json.dumps(response.model_dump(), indent=2, default=str))
```

The above code outputs:

```
{
  "id": "chatcmpl-CO29dOpaUINzLPmGsSouk2L9VgLHD",
  "choices": [
    {
      "finish_reason": "tool_calls",
      "index": 0,
      "logprobs": null,
      "message": {
        "content": null,
        "refusal": null,
        "role": "assistant",
        "annotations": [],
        "audio": null,
        "function_call": null,
        "tool_calls": [
          {
            "id": "call_5b7Vu6HrolUDSAajdaXSKIAc",
            "function": {
              "arguments": "{}",
              "name": "get_current_time"
            },
            "type": "function"
          }
        ]
      }
    }
  ],
  "created": 1759844189,
  "model": "gpt-4o-2024-08-06",
  "object": "chat.completion",
  "service_tier": "default",
  "system_fingerprint": "fp_cbf1785567",
```

```
  "usage": {
    "completion_tokens": 11,
    "prompt_tokens": 45,
    "total_tokens": 56,
    "completion_tokens_details": {
      "accepted_prediction_tokens": 0,
      "audio_tokens": 0,
      "reasoning_tokens": 0,
      "rejected_prediction_tokens": 0
    },
    "prompt_tokens_details": {
      "audio_tokens": 0,
      "cached_tokens": 0
    }
  }
}
```

Now invoke the tool selected above in LLM's response:

```
response2 = None

# Create a condition in case tool_calls is in response object
if response.choices[0].message.tool_calls:
    # Pull out the specific tool metadata from the response
    tool_call = response.choices[0].message.tool_calls[0]
    args = json.loads(tool_call.function.arguments)

    # Run the tool locally
    tool_result = get_current_time()

    # Append the result to the messages list
    messages.append(response.choices[0].message)
    messages.append({
        "role": "tool", "tool_call_id": tool_call.id, "content":
str(tool_result)
    })

    # Send the list of messages with the newly appended results
back to the LLM
    response2 = client.chat.completions.create(
        model="openai:gpt-4o",
        messages=messages,
        tools=tools,
    )

    print(response2.choices[0].message.content)
```

More tools to LLM

```
import requests
import qrcode
from qrcode.image.styledpil import StyledPilImage


def get_weather_from_ip():
    """
    Gets the current, high, and low temperature in Fahrenheit for
the user's
    location and returns it to the user.
    """
    # Get location coordinates from the IP address
    lat, lon = requests.get('https://ipinfo.io/json').json()
['loc'].split(',')

    # Set parameters for the weather API call
    params = {
        "latitude": lat,
        "longitude": lon,
        "current": "temperature_2m",
        "daily": "temperature_2m_max,temperature_2m_min",
        "temperature_unit": "fahrenheit",
        "timezone": "auto"
    }

    # Get weather data
    weather_data = requests.get("https://api.open-meteo.com/v1/
forecast", params=params).json()

    # Format and return the simplified string
    return (
        f"Current: {weather_data['current']['temperature_2m']}°F,
"
        f"High: {weather_data['daily']['temperature_2m_max'][0]}
°F, "
        f"Low: {weather_data['daily']['temperature_2m_min'][0]}
°F"
    )

# Write a text file
def write_txt_file(file_path: str, content: str):
    """
    Write a string into a .txt file (overwrites if exists).
    Args:
        file_path (str): Destination path.
        content (str): Text to write.
    Returns:
```

```
        str: Path to the written file.
    """
    with open(file_path, "w", encoding="utf-8") as f:
        f.write(content)
    return file_path


# Create a QR code
def generate_qr_code(data: str, filename: str, image_path: str):
    """Generate a QR code image given data and an image path.

    Args:
        data: Text or URL to encode
        filename: Name for the output PNG file (without
extension)
        image_path: Path to the image to be used in the QR code
    """
    qr =
qrcode.QRCode(error_correction=qrcode.constants.ERROR_CORRECT_H)
    qr.add_data(data)

    img = qr.make_image(image_factory=StyledPilImage,
embedded_image_path=image_path)
    output_file = f"{filename}.png"
    img.save(output_file)

    return f"QR code saved as {output_file} containing:
{data[:50]}..."
```

Using the new tools

```
prompt = "Can you get the weather for my location?"

response = client.chat.completions.create(
    model="openai:o4-mini",
    messages=[{"role": "user", "content": (
        prompt
    )}],
    tools=[
        get_current_time,
        get_weather_from_ip,
        write_txt_file,
        generate_qr_code
    ],
    max_turns=5
)

display_functions.pretty_print_chat_completion(response)
```

LLM Action: `get_weather_from_ip`
`{}`

Tool Response: `get_weather_from_ip`
`"Current: 61.6\u00b0F, High: 68.7\u00b0F, Low: 54.6\u00b0F"`

Final Assistant Message:
Here's the weather for your location: - Current temperature: 61.6°F - High today: 68.7°F - Low today: 54.6°F

LLM appropriately chose the correct tool based on the intent of the prompt, even though it had access to other tools.

Calling Multiple tools

```
prompt = "Can you help me create a qr code that goes to
www.deeplearning.com from the image dl_logo.jpg? Also write me a
txt note with the current weather please."

response = client.chat.completions.create(
    model="openai:o4-mini",
    messages=[{"role": "user", "content": (
        prompt
    )}],
    tools=[
        get_weather_from_ip,
        get_current_time,
        write_txt_file,
        generate_qr_code
    ],
    max_turns=10
)

display_functions.pretty_print_chat_completion(response)
```

🧠 **LLM Action:** `get_weather_from_ip`
`{}`
🔧 **Tool Response:** `get_weather_from_ip`
`"Current: 61.3\u00b0F, High: 68.7\u00b0F, Low: 54.6\u00b0F"`

🧠 **LLM Action:** `generate_qr_code`

```
{
  "data": "www.deeplearning.com",
  "filename": "deeplearning_qr",
  "image_path": "dl_logo.jpg"
}
```

🔧 **Tool Response:** `generate_qr_code`

```
"QR code saved as deeplearning_qr.png containing:
www.deeplearning.com..."
```

🧠 **LLM Action:** `write_txt_file`

```
{
  "file_path": "weather_note.txt",
  "content": "Current Weather: 61.3\u00b0F\nHigh:
68.7\u00b0F\nLow: 54.6\u00b0F"
}
```

🔧 **Tool Response:** `write_txt_file`

```
"weather_note.txt"
```

✅ **Final Assistant Message:**

I've created the QR code (saved as deeplearning_qr.png) that links to www.deeplearning.com with your dl_logo.jpg embedded, and wrote a text note (weather_note.txt) containing the current weather: Current Weather: 61.3°F High: 68.7°F Low: 54.6°F Let me know if you need anything else!

🧭 **Tool Sequence:**

get_weather_from_ip → generate_qr_code → write_txt_file

Best Practices for tool definition

- Keep tool **docstrings** short, imperative, and specific to the action.
- Return **consistent, compact JSON** so the model can chain results.
- Prefer **one clear responsibility per tool** (single route, single effect).

https://platform.openai.com/docs/guides/function-calling#best-practices-for-defining-functions

1. **Write clear and detailed function names, parameter descriptions, and instructions.**

Explicitly describe the purpose of the function and each parameter (and its format), and what the output represents.

Use the system prompt to describe when (and when not) to use each function. Generally, tell the model *exactly* what to do.

Include examples and edge cases, especially to rectify any recurring failures. (**Note:** Adding examples may hurt performance for reasoning models.)

2. **Apply software engineering best practices.**

   Make the functions obvious and intuitive. (principle of least surprise)

   Use enums and object structure to make invalid states unrepresentable. (e.g. `toggle_light(on: bool, off: bool)` allows for invalid calls)

   Pass the intern test. Can an intern/human correctly use the function given nothing but what you gave the model? (If not, what questions do they ask you? Add the answers to the prompt.)

3. **Offload the burden from the model and use code where possible.**

   Don't make the model fill arguments you already know. For example, if you already have an `order_id` based on a previous menu, don't have an `order_id` param – instead, have no params `submit_refund()` and pass the `order_id` with code.

   Combine functions that are always called in sequence. For example, if you always call `mark_location()` after `query_location()`, just move the marking logic into the query function call.

4. **Keep the number of functions small for higher accuracy.**

   Evaluate your performance with different numbers of functions.

Aim for fewer than 20 functions at any one time, though this is just a soft suggestion.

Example of tool calling

```
from openai import OpenAI
import json

client = OpenAI()

# 1. Define a list of callable tools for the model
tools = [
    {
        "type": "function",
        "name": "get_horoscope",
        "description": "Get today's horoscope for an astrological
sign.",
        "parameters": {
            "type": "object",
            "properties": {
                "sign": {
                    "type": "string",
                    "description": "An astrological sign like
Taurus or Aquarius",
                },
            },
            "required": ["sign"],
        },
    },
]

def get_horoscope(sign):
    return f"{sign}: Next Tuesday you will befriend a baby
otter."

# Create a running input list we will add to over time
input_list = [
    {"role": "user", "content": "What is my horoscope? I am an
Aquarius."}
]

# 2. Prompt the model with tools defined
response = client.responses.create(
    model="gpt-5",
    tools=tools,
    input=input_list,
)
```

```
# Save function call outputs for subsequent requests
input_list += response.output

for item in response.output:
    if item.type == "function_call":
        if item.name == "get_horoscope":
            # 3. Execute the function logic for get_horoscope
            horoscope = get_horoscope(json.loads(item.arguments))

            # 4. Provide function call results to the model
            input_list.append({
                "type": "function_call_output",
                "call_id": item.call_id,
                "output": json.dumps({
                  "horoscope": horoscope
                })
            })

print("Final input:")
print(input_list)

response = client.responses.create(
    model="gpt-5",
    instructions="Respond only with a horoscope generated by a
tool.",
    tools=tools,
    input=input_list,
)

# 5. The model should be able to give a response!
print("Final output:")
print(response.model_dump_json(indent=2))
print("\n" + response.output_text)
```

Handling Function calls

The response `output` array contains an entry with the `type` having a value of `function_call`. Each entry with a `call_id` (used later to submit the function result), `name`, and JSON-encoded `arguments`.

Sample LLM response with functions to be called:

```
[
    {
```

```
            "id": "fc_12345xyz",
            "call_id": "call_12345xyz",
            "type": "function_call",
            "name": "get_weather",
            "arguments": "{\"location\":\"Paris, France\"}"
        },
        {
            "id": "fc_67890abc",
            "call_id": "call_67890abc",
            "type": "function_call",
            "name": "get_weather",
            "arguments": "{\"location\":\"Bogotá, Colombia\"}"
        },
        {
            "id": "fc_99999def",
            "call_id": "call_99999def",
            "type": "function_call",
            "name": "send_email",
            "arguments": "{\"to\":\"bob@email.com\",\"body\":\"Hi
bob\"}"
        }
    ]
```

Execute function calls and append results:

```
    for tool_call in response.output:
        if tool_call.type != "function_call":
            continue

        name = tool_call.name
        args = json.loads(tool_call.arguments)

        result = call_function(name, args)
        input_messages.append({
            "type": "function_call_output",
            "call_id": tool_call.call_id,
            "output": str(result)
        })

    def call_function(name, args):
        if name == "get_weather":
            return get_weather(**args)
        if name == "send_email":
            return send_email(**args)
```

A result must be a string, but the format is up to you (JSON, error codes, plain text, etc.). The model will interpret that string as needed.

If your function has no return value (e.g. `send_email`), simply return a string to indicate success or failure. (e.g. `"success"`)

Tool choice

By default the model will determine when and how many tools to use. You can force specific behavior with the `tool_choice` parameter.
https://platform.openai.com/docs/guides/function-calling#tool-choice

Recording tool call results

- **Understanding the `ChatCompletionMessage`** object will help you access the required attributes to save the messages. An example of `ChatCompletionMessage` looks like this:

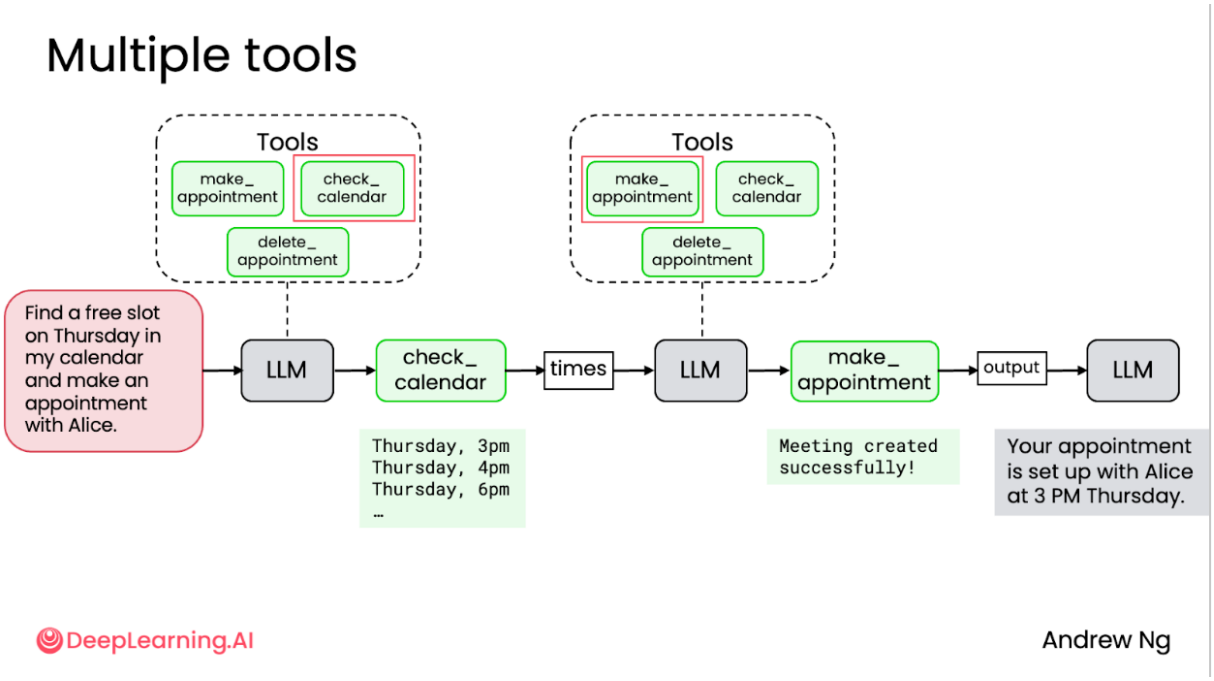
```
ChatCompletionMessage(
    content=None,
    refusal=None,
    role='assistant',
    annotations=[],
    audio=None,
    function_call=None,
    tool_calls=[
        ChatCompletionMessageFunctionToolCall(
            id='call_ymMki5TBB91efJhMPjgoqjop',
            function=Function(
                arguments='{"query":"radio observations of
recurrent novae","max_results":5}',
                name='arxiv_search_tool'
            ),
            type='function'
        )
    ]
)
```

Assuming that `msg` if of type `ChatCompletionMessage`, if you wanted to get the `name` of a `tool_call` you can do something like:

```
for call in msg.tool_calls:
    tool_name = call.function.name
```

Finally, the `result` variable will be created by actually calling the function associated with each tool (`tool_func`).

Project 2 - Email assistant workflow



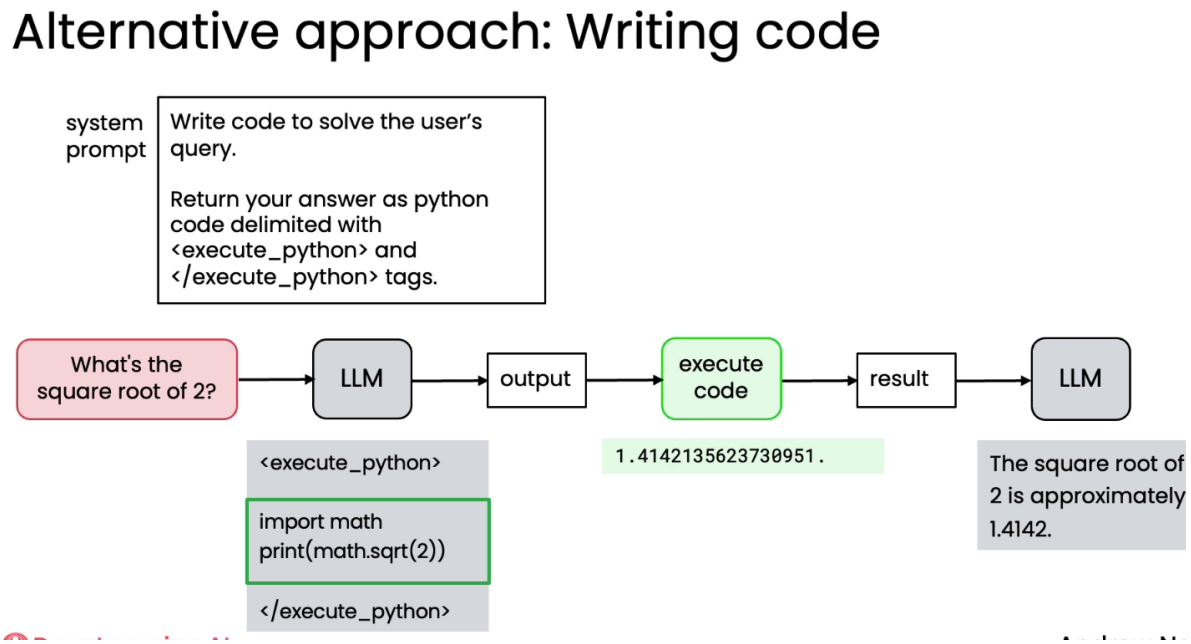
Available tools

Tool Function	Action
`list_all_emails()`	Fetch all emails, newest first
`list_unread_emails()`	Retrieve only unread emails
`search_emails(query)`	Search by keyword in subject, body, or sender
`filter_emails(...)`	Filter by recipient and/or date range
`get_email(email_id)`	Fetch a specific email by ID
`mark_email_as_read(id)`	Mark an email as read
`mark_email_as_unread(id)`	Mark an email as unread
`send_email(...)`	Send a new (mock) email

`delete_email(id)`	Delete an email by ID
`search_unread_from_sender(addr)`	Return unread emails from a given sender (e.g., `boss@email.com`)

```
def build_prompt(request_: str) -> str:
    return f"""
- You are an AI assistant specialized in managing emails.
- You can perform various actions such as listing, searching,
filtering, and manipulating emails.
- Use the provided tools to interact with the email system.
- Never ask the user for confirmation before performing an
action.
- If needed, my email address is "you@email.com" so you can use
it to send emails or perform actions related to my account.

{request_.strip()}
"""

example_prompt = build_prompt("Delete the Happy Hour email")
utils.print_html(content=example_prompt, title="Example
example_prompt")
```

Code execution



LLM generates the code which can then be executed.

Code execution needs to be done in secure sandbox environments like docker container or E2B sandbox service. Running LLM-generated code in a sandbox prevents accidental or malicious code from harming your system or leaking data.

MCP

Standard way of writing agent clients and tool servers.

Slack can provide a slack MCP server wrapping their APIs as MCP tools/ resources. An agentic client App that needs to use slack APIs will provide the Slack MCP tools to the LLM and once the LLM selects a certain slack tool to be called, the MCP client will invoke the corresponding tool on MCP server. Internally MCP service uses JSON-RPC protocol.

The main benefit is reducing redundant integration work by standardizing tool access, making LLM-based application development more efficient.

Project 3 - Agent to publish structured reports in HTML

A mini agent capable of searching, reasoning, and publishing structured reports in HTML—laying the foundation for more advanced multi-step and autonomous AI systems.

- Chain steps into a research pipeline (**search → reflection → formatting**).
- Convert natural-language output into **styled HTML** suitable for sharing.

Tools used:

- **`arxiv_search_tool(query, max_results)`** – academic papers via arXiv API.
- **`tavily_search_tool(query, max_results, include_images)`** – general web search via Tavily.

```
# ================================
# Standard library imports
# ================================
import json
```

```
# ================================
# Third-party imports
# ================================
from dotenv import load_dotenv
from openai import OpenAI
from IPython.display import display, HTML

# ================================
# Local / project imports
# ================================
import research_tools

# ================================
# Environment setup
# ================================
load_dotenv()  # Load environment variables from .env file

# Instantiate OpenAI's client (you should use this in your graded
functions)
CLIENT = OpenAI()
```

the `arxiv_search_tool` works.
This tool searches arXiv and returns a list of papers with:

- `title`, `authors`, `published`, `summary`, `url`, and (if available) `link_pdf`.

```
# Test the arXiv search tool
topic = "linear algebra"

arxiv_results = research_tools.arxiv_search_tool(topic,
max_results=3)

# Show formatted arxiv_results
for i, paper in enumerate(arxiv_results, 1):
    if "error" in paper:
        print(f"❌ Error: {paper['error']}")
    else:
        print(f"📄 Paper {i}")
        print(f"  Title     : {paper['title']}")
        print(f"  Authors   : {', '.join(paper['authors'])}")
        print(f"  Published : {paper['published']}")
        print(f"  URL       : {paper['url']}\n")
```

```
print("\n🧾 Raw arxiv_Results:\n")
print(json.dumps(arxiv_results, indent=2))
```

The `tavily_search_tool` calls the Tavily API to fetch web results. Returns a list of dicts:

- `title`, `content`, `url` (and optional image URLs when `include_images=True`).

```
# Test the Tavily search tool
topic = "retrieval-augmented generation applications"

tavily_results = research_tools.tavily_search_tool(topic)
for item in tavily_results:
    print(item)
```

a dictionary that maps tool names (strings) to the actual Python functions. This allows the model to call tools by name during tool-calling.

```
# Tool mapping
TOOL_MAPPING = {
    "tavily_search_tool": research_tools.tavily_search_tool,
    "arxiv_search_tool": research_tools.arxiv_search_tool,
}

def generate_research_report_with_tools(prompt: str, model: str =
"gpt-4o") -> str:
    """
    Generates a research report using OpenAI's tool-calling with
arXiv and Tavily tools.

    Args:
        prompt (str): The user prompt.
        model (str): OpenAI model name.

    Returns:
        str: Final assistant research report text.
    """
    messages = [
        {
            "role": "system",
            "content": (
                "You are a research assistant that can search the
web and arXiv to write detailed, "
                "accurate, and properly sourced research reports.
\n\n"
```

```
                "🔍 Use tools when appropriate (e.g., to find
scientific papers or web content).\n"
                "📚 Cite sources whenever relevant. Do NOT omit
citations for brevity.\n"
                "🌐 When possible, include full URLs (arXiv
links, web sources, etc.).\n"
                "✍️ Use an academic tone, organize output into
clearly labeled sections, and include "
                "inline citations or footnotes as needed.\n"
                "🚫 Do not include placeholder text such as
'(citation needed)' or '(citations omitted)'."
            )
        },
        {"role": "user", "content": prompt}
    ]

    # List of available tools
    tools = [research_tools.arxiv_tool_def,
research_tools.tavily_tool_def]

    # Maximum number of turns
    max_turns = 10

    # Iterate for max_turns iterations
    for _ in range(max_turns):

        ### START CODE HERE ###

        # Chat with the LLM via the client and set the correct
arguments. Hint: Their names match names of variables already
defined.
        # Make sure to let the LLM choose tools automatically.
Hint: Look at the docs provided earlier!
        response = CLIENT.chat.completions.create(
            model=model,
            messages=messages,
            tools=tools,
            tool_choice="auto",
            temperature=1
        )

        ### END CODE HERE ###

        # Get the response from the LLM and append to messages
        msg = response.choices[0].message
        messages.append(msg)
```

```
        # Stop when the assistant returns a final answer (no tool
calls)
        if not msg.tool_calls:
            final_text = msg.content
            print("✅ Final answer:")
            print(final_text)
            break

        # Execute tool calls and append results
        for call in msg.tool_calls:
            tool_name = call.function.name
            args = json.loads(call.function.arguments)
            print(f"🛠️ {tool_name}({args})")

            try:
                tool_func = TOOL_MAPPING[tool_name]
                result = tool_func(**args)
            except Exception as e:
                result = {"error": str(e)}

            ### START CODE HERE ###

            # Keep track of tool use in a new message
            new_msg = {
                # Set role to "tool" (plain string) to signal a
tool was used
                "role": "tool",
                # As stated in the markdown when inspecting the
ChatCompletionMessage object
                # every call has an attribute called id
                "tool_call_id": call.id,
                # The name of the tool was already defined above,
use that variable
                "name": tool_name,
                # Pass the result of calling the tool to
json.dumps
                "content": json.dumps(result)
            }

            ### END CODE HERE ###

            # Append to messages
            messages.append(new_msg)

    return final_text
```

Reflection + Rewrite

```
def reflection_and_rewrite(report, model: str = "gpt-4o-mini",
temperature: float = 0.3) -> dict:
    """
    Generates a structured reflection AND a revised research
report.
    Accepts raw text OR the messages list returned by
generate_research_report_with_tools.

    Returns:
        dict with keys:
          - "reflection": structured reflection text
          - "revised_report": improved version of the input
report
    """

    # Input can be plain text or a list of messages, this
function detects and parses accordingly
    report = research_tools.parse_input(report)

    ### START CODE HERE ###

    # Define the prompt. A multi-line f-string is typically used
for this.
    # Remember it should ask the model to output ONLY valid JSON
with this structure:
    # {{ "reflection": "<text>", "revised_report": "<text>" }}
    user_prompt = f"""Review the following report and provide a
structured reflection with the following sections:

Strengths

Limitations

Suggestions

Opportunities

After the reflection, rewrite the report to address the
limitations and incorporate suggestions and opportunities, making
the report more effective and insightful.

Return your output strictly in this JSON format:
{{
"reflection": "Strengths: <list or paragraph> Limitations: <list
or paragraph> Suggestions: <list or paragraph> Opportunities:
<list or paragraph>",
"revised_report": "<Insert your revised report here>"
}}
```

```
Do not include anything outside this JSON object.

    """

    # Get a response from the LLM
    response = CLIENT.chat.completions.create(
        # Pass in the model
        model=model,
        messages=[
            # System prompt is already defined
            {"role": "system", "content": "You are an academic
reviewer and editor."},
            # Add user prompt
            {"role": "user", "content": user_prompt},
        ],
        # Set the temperature equal to the temperature parameter
passed to the function
        temperature=temperature
    )

    ### END CODE HERE ###

    # Extract output
    llm_output = response.choices[0].message.content.strip()

    # Check if output is valid JSON
    try:
        data = json.loads(llm_output)
    except json.JSONDecodeError:
        raise Exception("The output of the LLM was not valid
JSON. Adjust your prompt.")

    return {
        "reflection": str(data.get("reflection", "")).strip(),
        "revised_report": str(data.get("revised_report",
"")).strip(),
    }
```

Convert report to HTML

```
def convert_report_to_html(report, model: str = "gpt-4o",
temperature: float = 0.5) -> str:
    """
    Converts a plaintext research report into a styled HTML page
using OpenAI.
    Accepts raw text OR the messages list from the tool-calling
step.
    """
```

```
    # Input can be plain text or a list of messages, this
function detects and parses accordingly
    report = research_tools.parse_input(report)

    # System prompt is already provided
    system_prompt = "You convert plaintext reports into full
clean HTML documents."

    ### START CODE HERE ###

    # Build the user prompt instructing the model to return ONLY
valid HTML
    user_prompt = f"""Convert the following plain text research
report into a well-structured HTML document.

Use appropriate HTML tags for headings, subheadings, paragraphs,
and lists as needed.

Add section headings for key parts such as Introduction, Methods,
Results, Discussion, and Conclusion (or as fits the report
content).

Ensure the document is properly indented and formatted for
readability.

Output only the HTML code without any explanation or commentary.

Plain text report:
{report}

    """

    # Call the LLM by interacting with the CLIENT.
    # Remember to set the correct values for the model, messages
(system and user prompts) and temperature
    response = CLIENT.chat.completions.create(
        # Pass in the model
        model=model,
        messages=[
            # System prompt is already defined
            {"role": "system", "content": system_prompt},
            # Add user prompt
            {"role": "user", "content": user_prompt},
        ],
        # Set the temperature equal to the temperature parameter
passed to the function
        temperature=temperature
    )
```

```
    ### END CODE HERE ###

    # Extract the HTML from the assistant message
    html = response.choices[0].message.content.strip()

    return html
```

End to end pipeline

1. Generate a research report (tools).
2. Reflect on the report.
3. Convert the report to HTML.

```
# 1) Research with tools
prompt_ = "Radio observations of recurrent novae"
preliminary_report = generate_research_report_with_tools(prompt_)
print("=== Research Report (preliminary) ===\n")
print(preliminary_report)

# 2) Reflection on the report (use the final TEXT to avoid
ambiguity)
reflection_text = reflection_and_rewrite(preliminary_report)   #
<-- pass text, not messages
print("=== Reflection on Report ===\n")
print(reflection_text['reflection'], "\n")
print("=== Revised Report ===\n")
print(reflection_text['revised_report'], "\n")


# 3) Convert the report to HTML (use the TEXT and correct
function name)
html = convert_report_to_html(reflection_text['revised_report'])

print("=== Generated HTML (preview) ===\n")
print((html or "")[:600], "\n... [truncated]\n")

# 4) Display full HTML
display(HTML(html))
```

Create an eval to measure date extraction

1. Manually extract due dates from 10-20 invoices

test invoice 1 — per example ground truth

"August 20, 2025" → "2025/08/20"

2. Specify output format of data in prompt

Format the due date as YYYY/MM/DD

3. Extract date from the LLM response using code

```
date_pattern = r'\d{4}/\d{2}/\d{2}'
extracted_date = re.findall(date_pattern, llm_respons
```

4. Compare LLM result to ground truth

```
if (extracted_date == actual_date):
    num_correct +=1
```

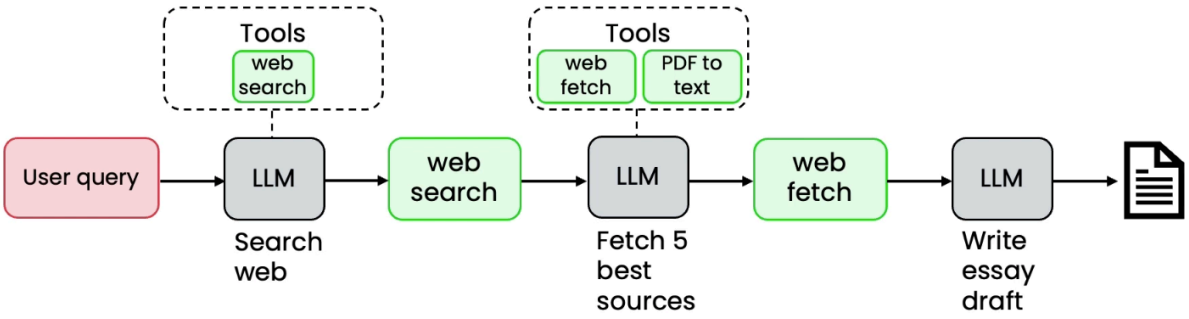
Driving your development process with evals

- Build a system and look at outputs to discover where it is behaving in an unsatisfactory way
  - E.g. incorrect due dates in invoice data extract
- Drive improvement by putting in place a small eval with ~20 examples to help you track progress
- Monitor as you make changes to workflow (e.g. new prompts, new algorithms) and see if the metric improves

Note:

1. Evaluate with all possible cases captured in the examples in evaluation dataset.

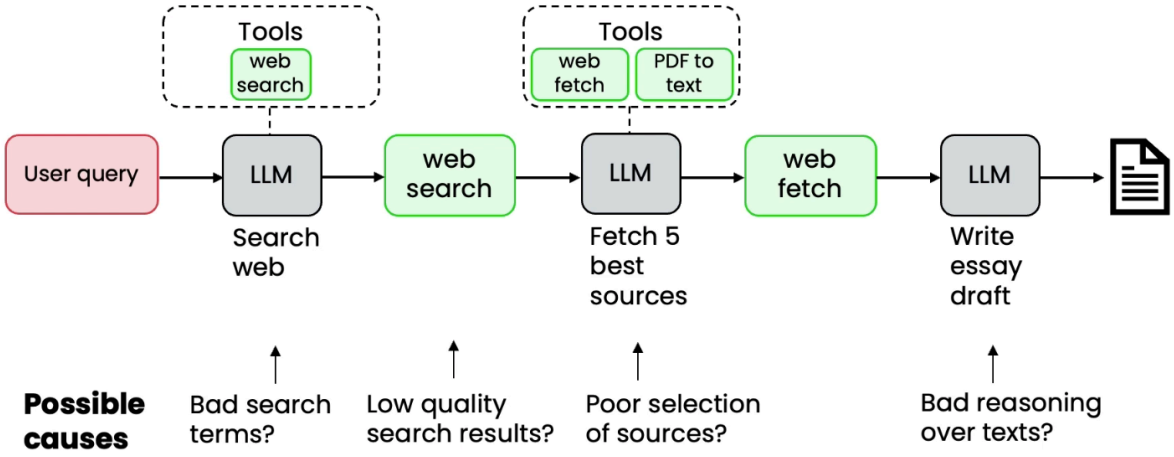
Example: research agent



Prompt	Issues
Recent black hole science	Missed high-profile result that had lots of news coverage
Renting vs buying a home in Seattle?	Seems to do a good job
Robotics for harvesting fruit	Didn't mention leading equipment company

Sometimes miss
points a human
would have mad

Example: research agent



Create an eval to measure performance

1. Choose 3-5 gold standard discussion points for each topic
2. Use LLM-as-a-judge to count how many topics were mentioned
3. Get score for each prompt in eval set

Example prompt	Gold-standard talking points
Black holes	Event horizon, radio telescope
Robotic harvesting	RoboPick, pinchers

ground truth annotations

Determine how many of the 5 gold-standard talking points are present in the provided essay.
Original Prompt
{original_prompt}
Essay to Evaluate
{essay_text}
Gold Standard Talking Points
{gold_standard_points}
Output Format
Return a json object with two keys: score (a single number between 0 and 5), and explanation (a string that lists the talking points present)

Subjective evals typically uses LLM as judge.

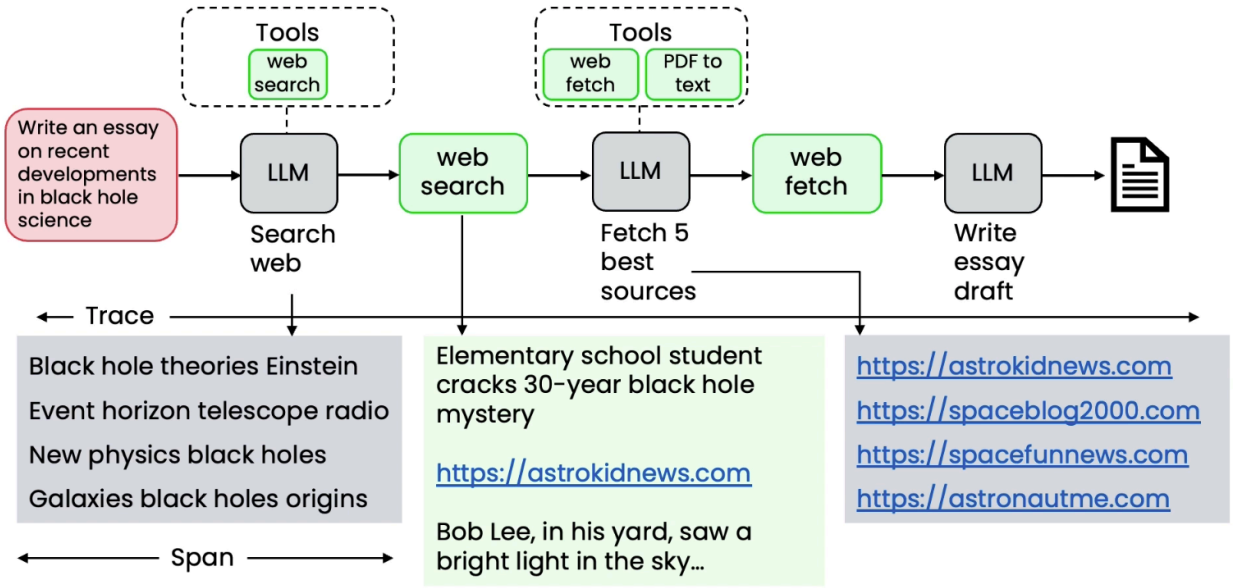
Two "axes" of evaluation

	Evaluate with code (objective)	LLM-as-judge (subjective)
Per example ground truth	Checking invoice date extraction `if (extracted_date == actual_date): num_correct +=1`	Counting gold-standard talking points Count the number of gold standard points in the following text…
No per example ground truth	Checking marketing copy length `if len(text) <= 10: num_correct += 1`	Grading charts with a rubric Grade this chart according to (i) whether it has clear axes labels, (ii) ….

Error Analysis and prioritizing next steps

1. Examine traces to better understand the workflow while trying to isolate the root cause for an issue. There could be several possible steps in the agentic workflow where the root cause of the issue may lie and traces will help isolate those.

Looking at traces



Counting up the errors

Prompt	Search terms	Search results	Picking 5 best sources	...	...
Recent developments in black hole science		Too many blog posts, not enough papers			
Renting vs buying a home in Seattle			Missed well-known blog		
Robotics for harvesting fruit	Terms too generic	Website for elementary school students			
...	...	...	...		
Batteries for electric vehicles		Only selected US-based companies	Missed magazine		
	5%	45%	10%	...	...

By analyzing note down where each root cause of issue with each eval query lies and create a spreadsheet with these.

Then we can focus on which component is seeing the most issues and we can prioritize on addressing that first. It could be by tweaking the prompts for web search tool call in the above example.

Tips for error analysis

- Develop a habit of looking at traces
- Carry out error analysis to figure out what component performed poorly, leading to a poor final output
- Use error analysis output to decide where to focus efforts

Counting up the errors

- Select 10-100 invoices for which the agentic workflow extracted the wrong due date

Input	PDF-to-text	LLM data extraction
Invoice 1	Errors in extraction	
Invoice 2		Wrong date selected
Invoice 3		Wrong data selected
...	...	...
Invoice 20	Errors in extraction	Wrong data selected
	15%	**87%**

Example: Responding to customer email

From: Susan Jones
Subject: Wrong item shipped

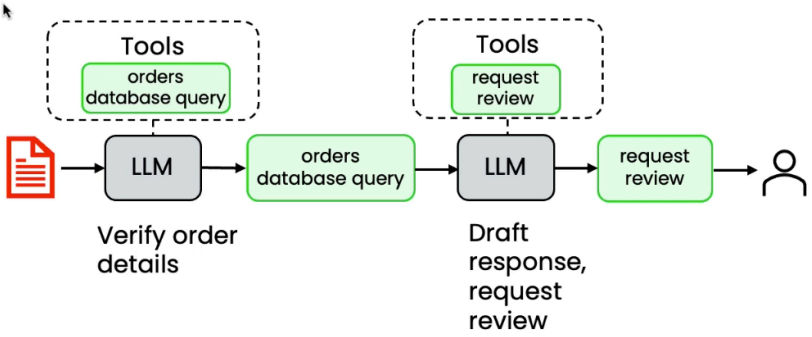
I ordered a blue KitchenPro blender (Order #8847) but received a red toaster instead.

I need the blender for my daughter's birthday party this weekend. Can you help?

Susan

Steps:

1. Extract key information
2. Find relevant customer records
3. Draft response for human review



Counting up the errors

Input	LLM-drafted query	Orders database query	LLM-drafted email
Email 1	Wrong table		
Email 2		Error in database entry	Didn't address details of order
Email 3	Incorrect math		
	...	...	...
Email 50			Defensive tone
	75%	4%	30%

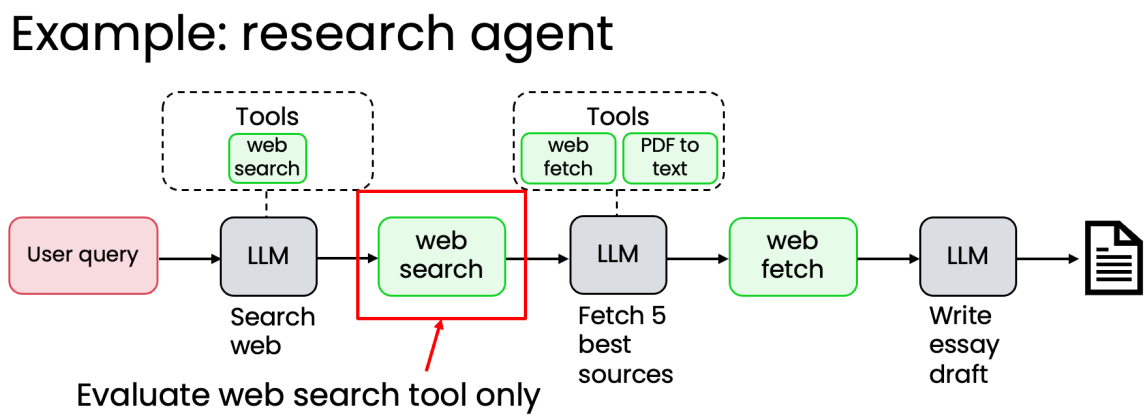
Component Level Evals

1. End to end eval is expensive
2. Component level evaluation can be a focussed evaluation of a component at a time. This avoids the noise in end-to-end system and focusses on making one component work better.

- If the problem lies in web search (usually the **first step** in a graded lab workflow), rerunning the *entire* pipeline (search → draft → reflect) every time can be **expensive** and noisy.
- Small improvements in web search quality may be hidden by randomness introduced by later components.
- By evaluating the web search *alone*, you get a **clearer signal** of whether that component is improving.

Component-level evals are also efficient when multiple teams are working on different pieces of a system: each team can optimize its own component using a clear metric, without needing to run or wait for full end-to-end tests.

Project - Component Level Evaluation to research workflow



- Create a list of gold standard web resources
- Write code that calculates how many results correspond to gold standard websites e.g. F1-score
- Track as you vary hyperparameters: e.g., search engine, number of results, dates

Steps in the research agent workflow:

1 Search the web for information.
2 Reflect on its output.
3 Publish a clear HTML report.

We will focus on step 1 - the research step.
The evaluation will compare the URLs retrieved by the agent against a **predefined list of preferred domains** (e.g., `arxiv.org`, `nature.com`, `nasa.gov`).
This allows you to quantify whether the system is pulling information from trustworthy sources, using an **objective, per-example ground truth evaluation**.
you will evaluate the reliability of sources by comparing them against a **predefined list of preferred domains**.
For this evaluation, we'll focus on the topic *"recent developments in black hole science"*, one of the examples highlighted in the course.
The idea is to verify whether the web search tool is returning sources from preferred domains, and to quantify the ratio of preferred vs. total results.

To build the eval, you will:

1. Extract the URLs returned by Tavily.
2. Compare them against a predefined list of **preferred domains** (e.g., `arxiv.org`, `nature.com`, `nasa.gov`).
3. Compute the **ratio of preferred vs. total results**.
4. Return a **PASS/FAIL flag** along with a Markdown-formatted summary.

```
# ========================
# Imports
# ========================

# --- Standard library
from datetime import datetime
import json
import re

# --- Third-party ---
from aisuite import Client

# --- Local / project ---
import research_tools
import utils

client = Client()

def find_references(task: str, model: str = "openai:gpt-4o",
```

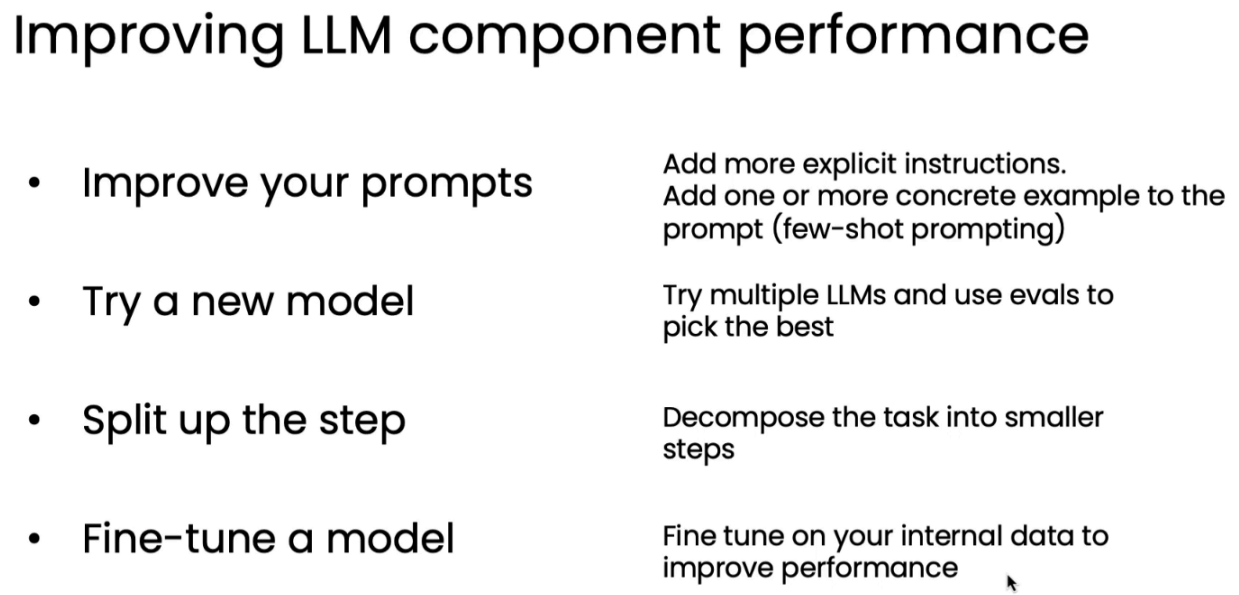
```
return_messages: bool = False):
    """Perform a research task using external tools (arxiv,
tavily, wikipedia)."""

    prompt = f"""
    You are a research function with access to:
    - arxiv_tool: academic papers
    - tavily_tool: general web search (return JSON when asked)
    - wikipedia_tool: encyclopedic summaries

    Task:
    {task}

    Today is {datetime.now().strftime('%Y-%m-%d')}.
    """.strip()

    messages = [{"role": "user", "content": prompt}]
    tools = [
        research_tools.arxiv_search_tool,
        research_tools.tavily_search_tool,
        research_tools.wikipedia_search_tool,
    ]

    try:
        response = client.chat.completions.create(
            model=model,
            messages=messages,
            tools=tools,
            tool_choice="auto",
            max_turns=5,
        )
        content = response.choices[0].message.content
        return (content, messages) if return_messages else
content
    except Exception as e:
        return f"[Model Error: {e}]"

research_task = "Find 2 recent papers about recent developments
in black hole science"
research_result = find_references(research_task)

utils.print_html(
    research_result,
    title="Research Function Output"
)

# list of preferred domains for Tavily results
TOP_DOMAINS = {
    # General reference / institutions / publishers
```

```
    "wikipedia.org", "nature.com", "science.org",
"sciencemag.org", "cell.com",
    "mit.edu", "stanford.edu", "harvard.edu", "nasa.gov",
"noaa.gov", "europa.eu",

    # CS/AI venues & indexes
    "arxiv.org", "acm.org", "ieee.org", "neurips.cc", "icml.cc",
"openreview.net",

    # Other reputable outlets
    "elifesciences.org", "pnas.org", "jmlr.org", "springer.com",
"sciencedirect.com",

    # Extra domains (case-specific additions)
    "pbs.org", "nova.edu", "nvcc.edu", "cccco.edu",

    # Well known programming sites
    "codecademy.com", "datacamp.com"
}

def evaluate_tavily_results(TOP_DOMAINS, raw: str,
min_ratio=0.4):
    """
    Evaluate whether plain-text research results mostly come from
preferred domains.

    Args:
        TOP_DOMAINS (set[str]): Set of preferred domains (e.g.,
'arxiv.org', 'nature.com').
        raw (str): Plain text or Markdown containing URLs.
        min_ratio (float): Minimum preferred ratio required to
pass (e.g., 0.4 = 40%).

    Returns:
        tuple[bool, str]: (flag, markdown_report)
            flag -> True if PASS, False if FAIL
            markdown_report -> Markdown-formatted summary of the
evaluation
    """

    # Extract URLs from the text
    url_pattern = re.compile(r'https?://[^\s\]\)>\}]+',
flags=re.IGNORECASE)
    urls = url_pattern.findall(raw)

    if not urls:
        return False, """### Evaluation — Tavily Preferred
Domains
No URLs detected in the provided text.
```

```
Please include links in your research results.
"""

    # Count preferred vs total
    total = len(urls)
    preferred_count = 0
    details = []

    for url in urls:
        domain = url.split("/")[2]
        preferred = any(td in domain for td in TOP_DOMAINS)
        if preferred:
            preferred_count += 1
        details.append(f"- {url} → {'✅ PREFERRED' if preferred
else '❌ NOT PREFERRED'}")

    ratio = preferred_count / total if total > 0 else 0.0
    flag = ratio >= min_ratio

    # Markdown report
    report = f"""
### Evaluation — Tavily Preferred Domains
- Total results: {total}
- Preferred results: {preferred_count}
- Ratio: {ratio:.2%}
- Threshold: {min_ratio:.0%}
- Status: {"✅ PASS" if flag else "❌ FAIL"}

**Details:**
{chr(10).join(details)}
"""
    return flag, report

# === 5.1. Try it yourself: topic, ratio & preferred domains ===
# Edit these parameters before running the cell

topic = "recent developments in black hole science"   # <- Change
the topic here
min_ratio = 0.4                                       # <- Change
threshold (0.0–1.0)
run_reflection = True                                 # <- Set
False to skip Step 4

# Short list of preferred domains (edit or expand as needed)
TOP_DOMAINS = {
    "wikipedia.org", "nature.com", "science.org", "arxiv.org",
    "nasa.gov", "mit.edu", "stanford.edu", "harvard.edu"
}
```

```
# Show a sample of preferred domains
import json
utils.print_html(
    json.dumps(sorted(list(TOP_DOMAINS)), indent=2),
    title="<h3>Sample Preferred Domains</h3>"
)

# 1) Research
research_task = f"Find 2–3 key papers and reliable overviews
about {topic}."
research_output = find_references(research_task)
utils.print_html(research_output, title=f"<h3>Research Results on
{topic}</h3>")

# 2) Evaluate sources (preferred domains ratio)
flag, eval_md = evaluate_tavily_results(TOP_DOMAINS,
research_output, min_ratio=min_ratio)
utils.print_html("<pre>" + eval_md + "</pre>",
title="<h3>Evaluation Summary</h3>")
```

How to address the problems identified



PII Removal Example

Instruction following

Summary of customer call:

On July 14, 2023, Jessica Alvarez (SSN: 555-44-3333) of 1024 Maple Ridge Lane, Boulder, CO 80301, submitted a support ticket....

Prompt

> Identify all cases of personally identifiable information (PII) in the text below.
>
> Then return a list of the identified PII classified by type, and then redact all the identified PII with "*****".
>
> Separate the list and the redacted text with "REDACTED: ".
> {text}

- Smaller models may not good at instruction following.
- **A larger model can identify all PIIs and do a better job at redaction.**

Instruction following results (GPT-5)

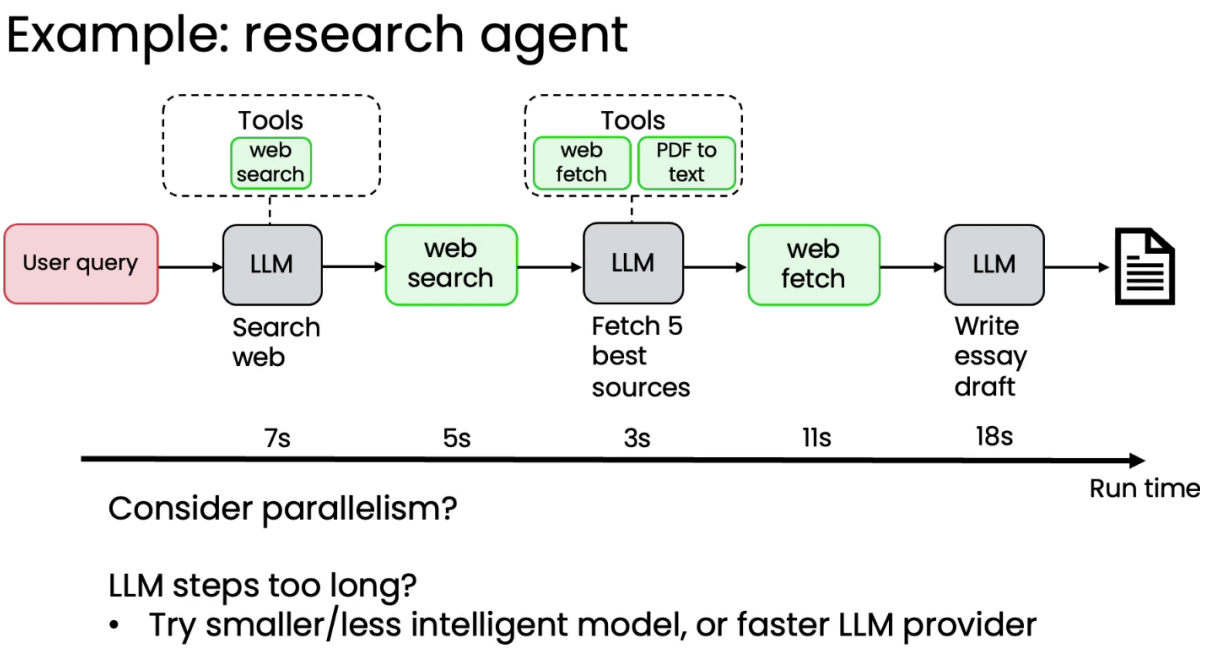
Identified PII (type → value):
1. Full Name → Jessica T. Alvarez
2. Social Security Number → 524-18-7629
3. Physical Address → 1024 Maple Ridge Lane, Boulder, CO 80301

REDACTED:
On July 14, 2023, ***** (SSN: *****), residing at *****, submitted a support ticket...

> Followed formatting instruction
>
> Identified all PII

Read other people's prompts helps understand prompting for a particular model works.

Latency and Cost Optimization

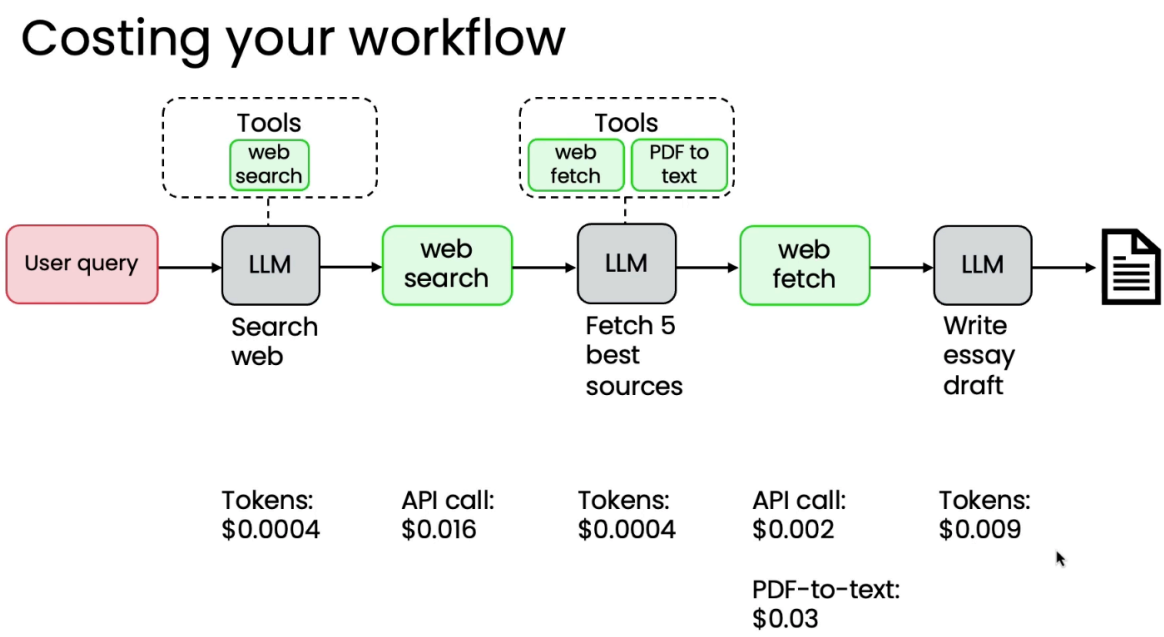


Benchmarking exercise for latency in workflow.

Analyze latency at each component

Analyze cost for agentic workflows:

1. LLM Steps (pay per token)
2. Any API-calling tools (pay per API call)
3. Compute steps (based on server capacity/cost)



Benchmarking exercise for costing the workflow.

Development process summary

1. Build - coding
   - build end-to-end system
   - improve individual component
2. Analyzing the system - what to improve to make the system more optimal
   - evals with compute metrics for end-to-end system
   - error analysis
   - examine outputs; traces;
   - component level evals

Planning workflows

Harder to control as LLM decides what plan to come up with.

Planning example: Customer service agent

Inventory Database

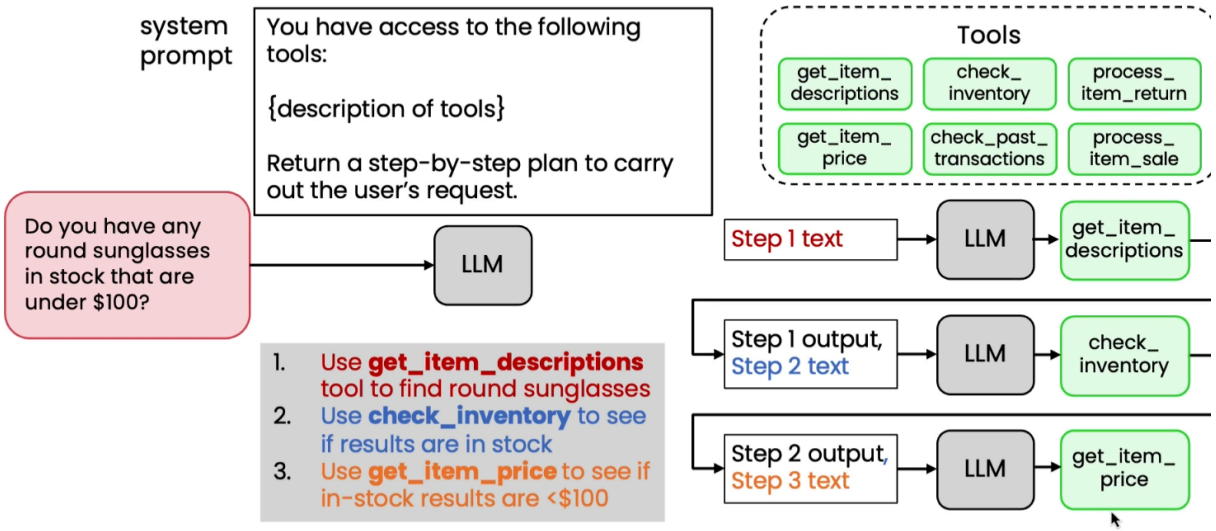
id	name	description	price	stock
1001	Aviator	Timeless pilot style for any occasion, metal frame	80	12
1002	Catseye	Glamorous 1950s profile, plastic frame	60	28
1003	Moon	Oversized round style, plastic frame	120	15
1004	Classic	Classic round profile, gold frame	60	9

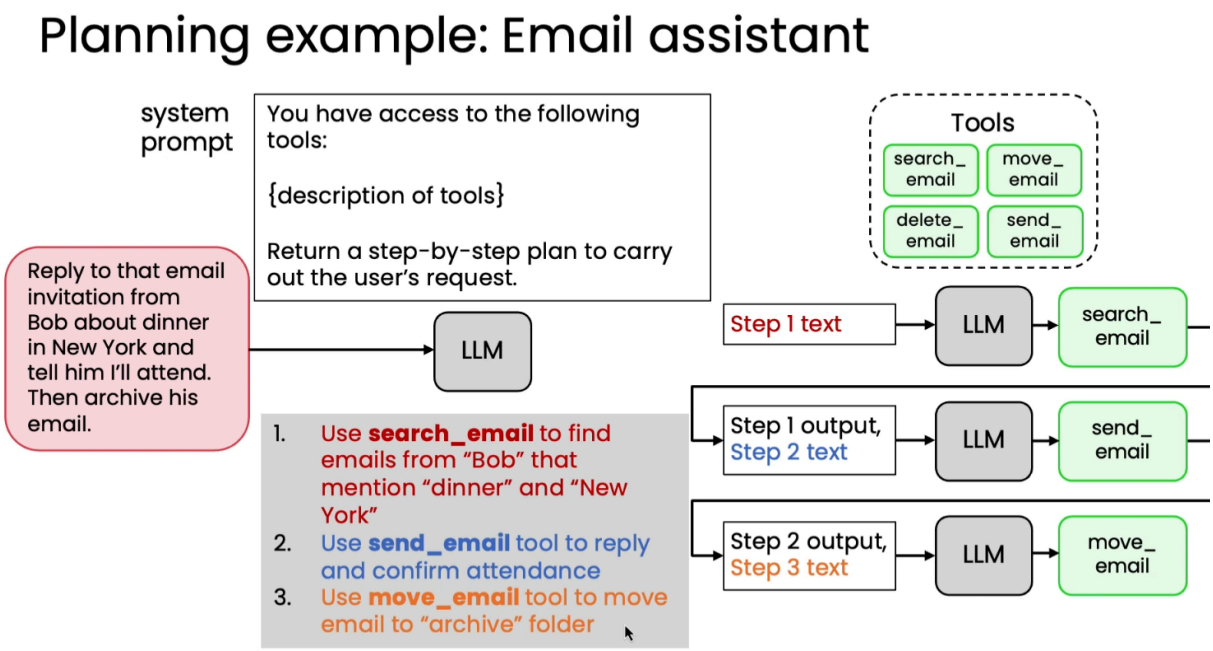
Customer query:

Do you have any round sunglasses in stock that are under $100?

Yes, we have our **Classic** sunglasses, which are a classic round metal frame and cost $60

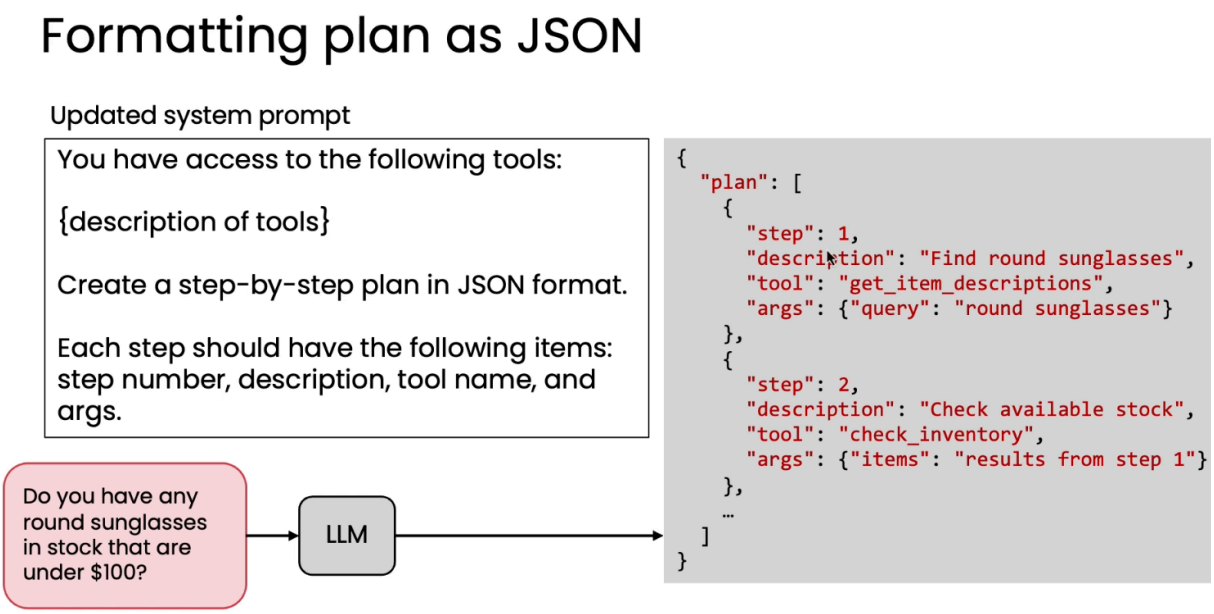
Planning example: Customer service agent





Creating and executing LLM Plans

Formatting plan as JSON makes it easy to parse the steps out from the plan.

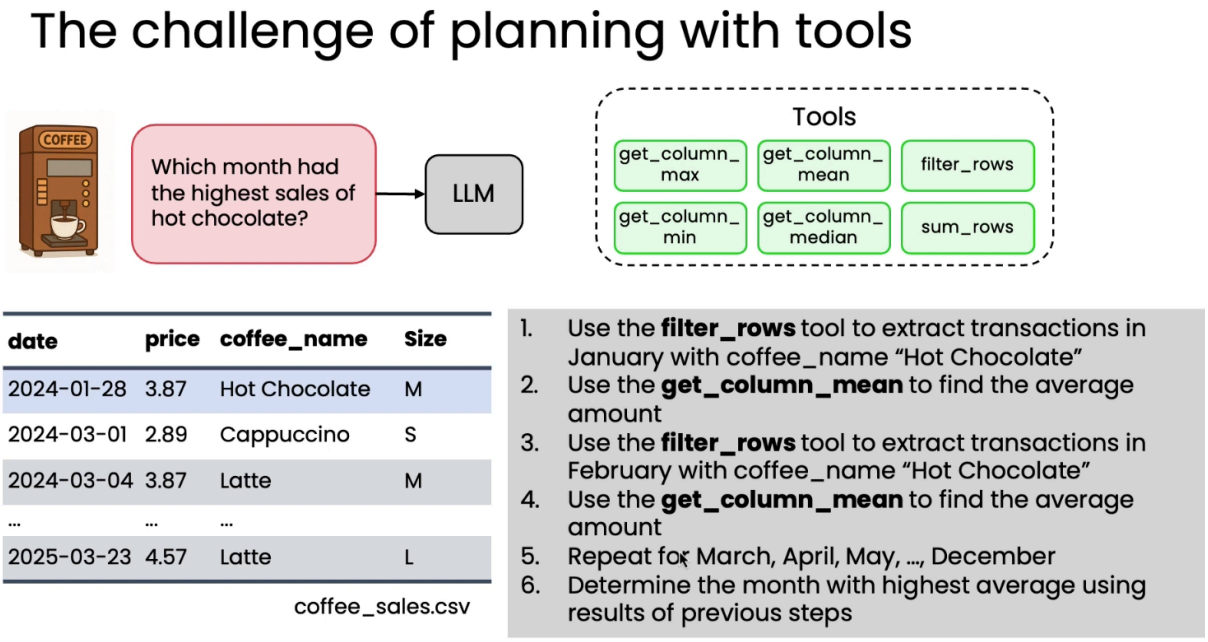


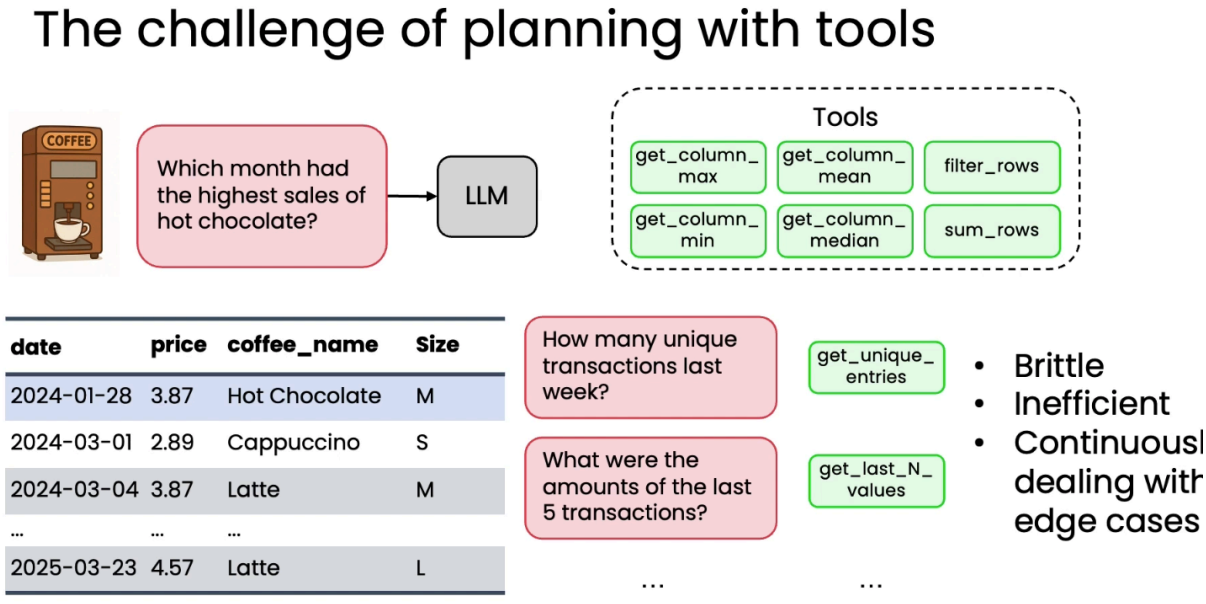
Planning with code execution

Executing code generated by LLM we can have more complex plans.

Letting an LLM generate and execute code enables flexible, complex planning without needing to predefine specific tools for

every task.
Increased flexibility allows broader task coverage but reduces predictability and control over agent actions and outcomes





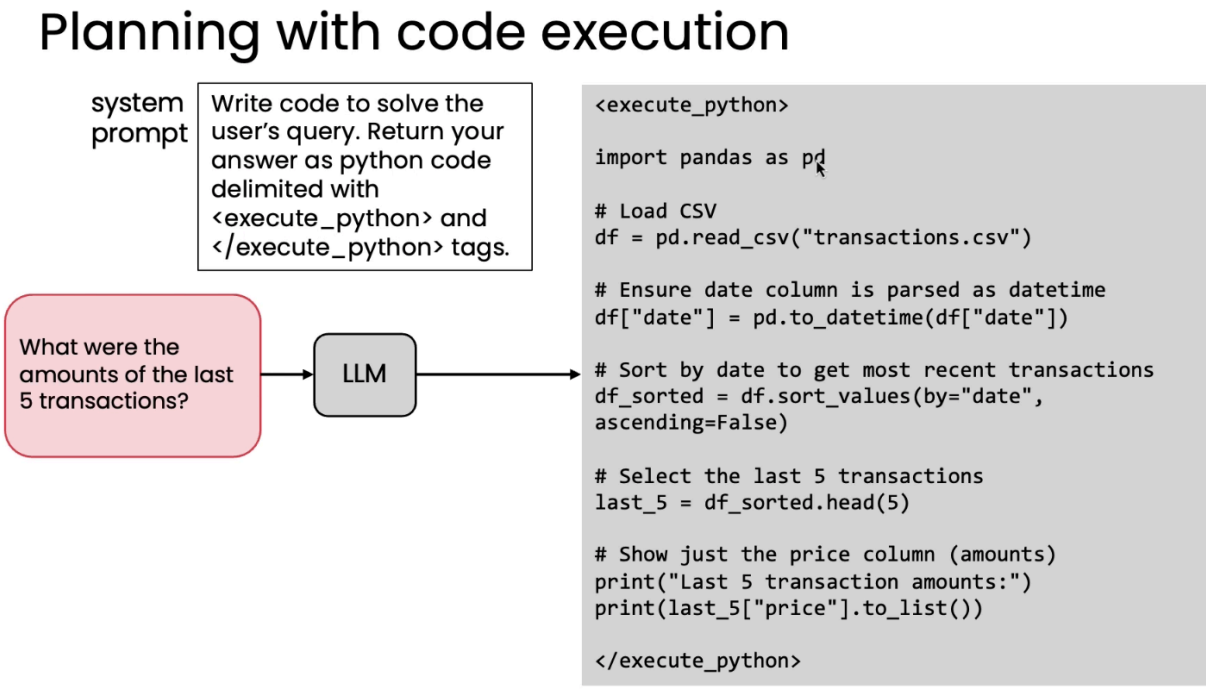
For various questions we may not have appropriate tools available and adding more tools will become a pain for every new use case.

So the solution is to have the LLM generate code with the planned steps. Get the data as pandas dataframe and then use pandas library (which has 100s of

functions in the library that LLM has seen during its training) to come up with the right data processing on the application side. So only few tools are typically needed to fetch the data and LLM will be able to generate the code using pandas library that can crunch over the fetched data to answer the query.

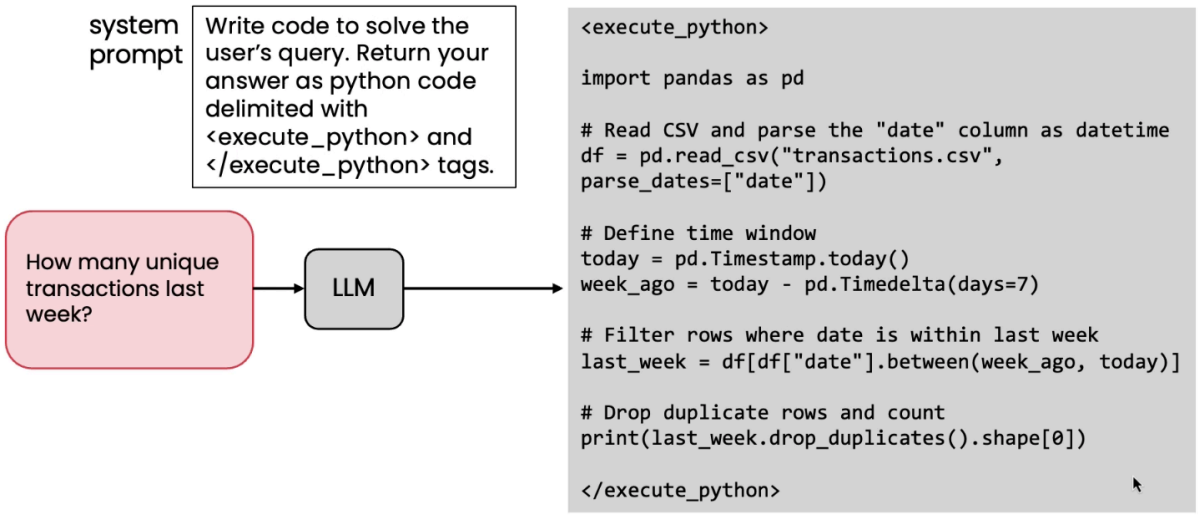
So instead of calling tools for each step in the plan, LLM can generate code which can carry out the steps in the plan using the data fetched using the tool call once with pandas library.

Code execution gives access to hundreds of built-in functions from libraries like Pandas, avoiding the need to create custom tools for every possible data query.

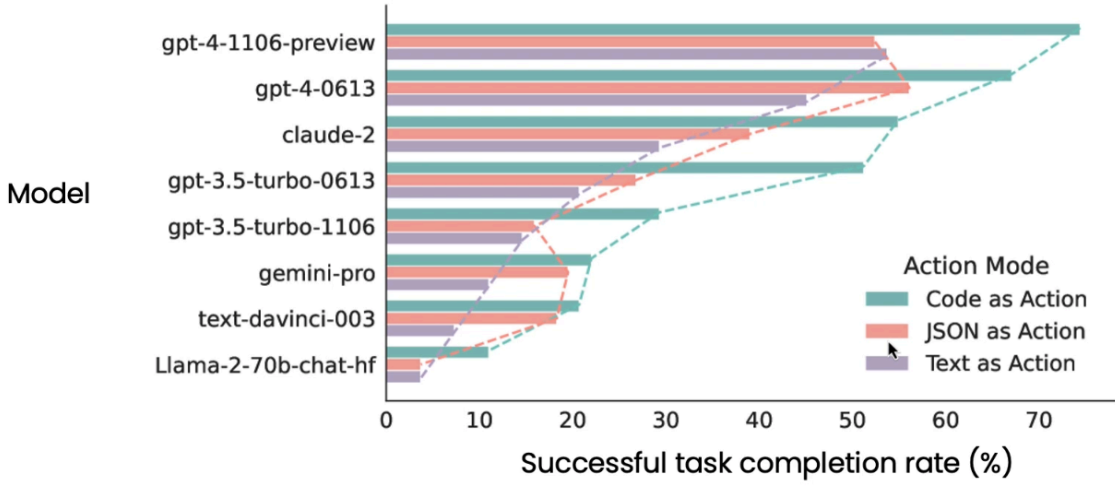


In such a case, the tool calling is one thing, the LLM generated code may too need to be evaluated.

Planning with code execution



Planning with code improves performance



[Adapted from "Executable Code actions Elicit Better LLM Agents", Wang et al. 2024]

Project 1 - Customer Service Agent

We will:

1. Create simple **inventory** and **transaction** datasets.
2. Build a **schema block** describing the data.
3. Prompt the LLM to **write a plan as Python code** (with comments explaining each step).
4. Execute the code in a sandbox to obtain the answer.

Inventory table


```
[
  {
    "item_id": "SG001",
    "name": "Aviator",
    "description": "Originally designed for pilots, these
teardrop-shaped lenses with thin metal frames offer timeless
appeal. The large lenses provide excellent coverage while the
lightweight construction ensures comfort during long wear.",
    "quantity_in_stock": 23,
    "price": 80
  },
  {
    "item_id": "SG002",
    "name": "Wayfarer",
    "description": "Featuring thick, angular frames that make a
statement, these sunglasses combine retro charm with modern edge.
The rectangular lenses and sturdy acetate construction create a
confident look.",
    "quantity_in_stock": 6,
    "price": 95
  },
  {
    "item_id": "SG003",
    "name": "Mystique",
    "description": "Inspired by 1950s glamour, these frames sweep
upward at the outer corners to create an elegant, feminine
silhouette. The subtle curves and often embellished temples add
sophistication to any outfit.",
    "quantity_in_stock": 3,
    "price": 70
  },
  {
    "item_id": "SG004",
    "name": "Sport",
    "description": "Designed for active lifestyles, these
wraparound sunglasses feature a single curved lens that provides
maximum coverage and wind protection. The lightweight, flexible
frames include rubber grips.",
    "quantity_in_stock": 11,
    "price": 110
  },
  {
    "item_id": "SG005",
    "name": "Classic",
    "description": "Classic round profile with minimalist metal
frames, offering a timeless and versatile style that fits both
```

```
casual and formal wear.",
    "quantity_in_stock": 10,
    "price": 60
  },
  {
    "item_id": "SG006",
    "name": "Moon",
    "description": "Oversized round style with bold plastic
frames, evoking retro aesthetics with a modern twist.",
    "quantity_in_stock": 10,
    "price": 120
  }
]
```

Transaction table

```
[
  {
    "transaction_id": "TXN001",
    "customer_name": "OPENING_BALANCE",
    "transaction_summary": "Daily opening register balance",
    "transaction_amount": 500.0,
    "balance_after_transaction": 500.0,
    "timestamp": "2025-10-11T18:00:09.253650"
  }
]
```

The plan

the model write Python that chains the steps (filter → compute → update) and then you just ran it.

Instead of piling on tiny tools or JSON plans, you used Python/TinyDB—giving the model a big, familiar toolbox that handles many query shapes with one prompt.

```
# ==== Imports ====
from __future__ import annotations
import json
from dotenv import load_dotenv
from openai import OpenAI
import re, io, sys, traceback, json
from typing import Any, Dict, Optional
from tinydb import Query, where

# Utility modules
```

```
import utils      # helper functions for prompting/printing
import inv_utils  # functions for inventory, transactions, schema
building, and TinyDB seeding

load_dotenv()
client = OpenAI()

db, inventory_tbl, transactions_tbl = inv_utils.seed_db()



PROMPT = """You are a senior data assistant. PLAN BY WRITING
PYTHON CODE USING TINYDB.

Database Schema & Samples (read-only):
{schema_block}

Execution Environment (already imported/provided):
- Variables: db, inventory_tbl, transactions_tbl  # TinyDB Table
objects
- Helpers: get_current_balance(tbl) -> float,
next_transaction_id(tbl, prefix="TXN") -> str
- Natural language: user_request: str  # the original user
message

PLANNING RULES (critical):
- Derive ALL filters/parameters from user_request (shape/
keywords, price ranges "under/over/between", stock mentions,
  quantities, buy/return intent). Do NOT hard-code values.
- Build TinyDB queries dynamically with Query(). If a constraint
isn't in user_request, don't apply it.
- Be conservative: if intent is ambiguous, do read-only (DRY
RUN).

TRANSACTION POLICY (hard):
- Do NOT create aggregated multi-item transactions.
- If the request contains multiple items, create a separate
transaction row PER ITEM.
- For each item:
  - compute its own line total (unit_price * qty),
  - insert ONE transaction with that amount,
  - update balance sequentially (balance += line_total),
  - update the item's stock.
- If any requested item lacks sufficient stock, do NOT mutate
anything; reply with STATUS="insufficient_stock".

HUMAN RESPONSE REQUIREMENT (hard):
- You MUST set a variable named `answer_text` (type str) with a
```

```
short, customer-friendly sentence (1–2 lines).
- This sentence is the only user-facing message. No dataframes/
JSON, no boilerplate disclaimers.
- If nothing matches, politely say so and offer a nearby
alternative (closest style/price) or a next step.

ACTION POLICY:
- If the request clearly asks to change state (buy/purchase/
return/restock/adjust):
    ACTION="mutate"; SHOULD_MUTATE=True; perform the change and
write a matching transaction row.
  Otherwise:
    ACTION="read"; SHOULD_MUTATE=False; simulate and explain
briefly as a dry run (in logs only).

FAILURE & EDGE-CASE HANDLING (must implement):
- Do not capture outer variables in Query.test. Pass them as
explicit args.
- Always set a short `answer_text`. Also set a string `STATUS` to
one of:
  "success", "no_match", "insufficient_stock", "invalid_request",
"unsupported_intent".
- no_match: No items satisfy the filters → suggest the closest in
style/price, or invite a different range.
- insufficient_stock: Item found but stock < requested qty →
state available qty and offer the max you can fulfill.
- invalid_request: Unable to parse essential info (e.g., quantity
for a purchase/return) → ask for the missing piece succinctly.
- unsupported_intent: The action is outside the store's
capabilities → provide the nearest supported alternative.
- In all cases, keep the tone helpful and concise (1–2
sentences). Put technical details (e.g., ACTION/DRY RUN) only in
stdout logs.

OUTPUT CONTRACT:
- Return ONLY executable Python between these tags (no extra
text):
  <execute_python>
  # your python
  </execute_python>

CODE CHECKLIST (follow in code):
1) Parse intent & constraints from user_request (regex ok).
2) Build TinyDB condition incrementally; query inventory_tbl.
3) If mutate: validate stock, update inventory, insert a
transaction (new id, amount, balance, timestamp).
4) ALWAYS set:
   - `answer_text` (human sentence, required),
   - `STATUS` (see list above).
```

```
   Also print a brief log to stdout, e.g., "LOG: ACTION=read
DRY_RUN=True STATUS=no_match".
5) Optional: set `answer_rows` or `answer_json` if useful, but
`answer_text` is mandatory.

TONE EXAMPLES (for `answer_text`):
- success: "Yes, we have our Classic sunglasses, a round frame,
for $60."
- no_match: "We don’t have round frames under $100 in stock right
now, but our Moon round frame is available at $120."
- insufficient_stock: "We only have 1 pair of Classic left; I can
reserve that for you."
- invalid_request: "I can help with that—how many pairs would you
like to purchase?"
- unsupported_intent: "We can’t refurbish frames, but I can
suggest similar new models."

Constraints:
- Use TinyDB Query for filtering. Standard library imports only
if needed.
- Keep code clear and commented with numbered steps.

User request:
{question}
"""

# ---------- 1) Code generation ----------
def generate_llm_code(
    prompt: str,
    *,
    inventory_tbl,
    transactions_tbl,
    model: str = "gpt-4.1-mini",
    temperature: float = 0.2,
) -> str:
    """
    Ask the LLM to produce a plan-with-code response.
    Returns the FULL assistant content (including surrounding
text and tags).
    The actual code extraction happens later in
execute_generated_code.
    """
    schema_block = inv_utils.build_schema_block(inventory_tbl,
transactions_tbl)
    prompt = PROMPT.format(schema_block=schema_block,
question=prompt)

    resp = client.chat.completions.create(
        model=model,
```

```
        temperature=temperature,
        messages=[
            {
                "role": "system",
                "content": "You write safe, well-commented TinyDB
code to handle data questions and updates."
            },
            {"role": "user", "content": prompt},
        ],
    )
    content = resp.choices[0].message.content or ""

    return content

prompt_round = "Do you have any round sunglasses in stock that
are under $100?"

# Generate the plan-as-code (FULL content; may include
<execute_python> tags)
full_content_round = generate_llm_code(
    prompt_round,
    inventory_tbl=inventory_tbl,
    transactions_tbl=transactions_tbl,
    model="o4-mini",
    temperature=1.0,
)

# Inspect the LLM's plan + code (no execution here)
utils.print_html(full_content_round, title="Plan with Code (Full
Response)")
```

LLM Response

```
<execute_python>
# 1) Import required modules
import re
from tinydb import Query
from functools import reduce
from operator import and_

# 2) Parse filters from user_request
style_keyword = None
price_max = None

# Look for style keyword "round"
if re.search(r'\bround\b', user_request, re.IGNORECASE):
    style_keyword = 'round'

# Look for price constraint "under $X"
```

```
m_price = re.search(r'under\s*\$?(\d+)', user_request,
re.IGNORECASE)
if m_price:
    price_max = float(m_price.group(1))

# 3) Build TinyDB query conditions
conds = []
if style_keyword:
    # Match 'round' in description or name
    conds.append(Query().description.test(lambda d,
kw=style_keyword: kw.lower() in d.lower()))
conds.append(Query().quantity_in_stock > 0)  # ensure in stock
if price_max is not None:
    conds.append(Query().price < price_max)

# Combine conditions with AND
if conds:
    final_cond = reduce(and_, conds)
else:
    final_cond = Query().exists()  # no filter

# 4) Execute read-only search
matched = inventory_tbl.search(final_cond)

# 5) Prepare response based on results
if matched:
    # Choose the lowest-priced match
    best = min(matched, key=lambda item: item['price'])
    answer_text = f"Yes, we have our {best['name']} round
sunglasses for ${best['price']}."
    STATUS = "success"
else:
    # No exact match: suggest closest round style
    all_round =
inventory_tbl.search(Query().description.test(lambda d: 'round'
in d.lower()))
    if all_round and price_max is not None:
        # find the cheapest above the max price
        over = [it for it in all_round if it['price'] >=
price_max]
        if over:
            closest = min(over, key=lambda it: it['price'])
            answer_text = (f"We don’t have round frames under $
{int(price_max)} in stock right now, "
                           f"but our {closest['name']} round
frame is available at ${closest['price']}.")
        else:
            answer_text = f"We don’t have round frames under $
{int(price_max)} in stock right now."
```

```
    else:
        answer_text = "I’m sorry, we don’t carry round sunglasses
at the moment."
    STATUS = "no_match"

# 6) Log the operation
print(f"LOG: ACTION=read DRY_RUN=True STATUS={STATUS}")
</execute_python>
```

Define the executor function to run the given plan

```
# --- Helper: extract code between <execute_python>...</
execute_python> ---
def _extract_execute_block(text: str) -> str:
    """
    Returns the Python code inside <execute_python>...</
execute_python>.
    If no tags are found, assumes 'text' is already raw Python
code.
    """
    if not text:
        raise RuntimeError("Empty content passed to code
executor.")
    m = re.search(r"<execute_python>(.*?)</execute_python>",
text, re.DOTALL | re.IGNORECASE)
    return m.group(1).strip() if m else text.strip()


# ---------- 2) Code execution ----------
def execute_generated_code(
    code_or_content: str,
    *,
    db,
    inventory_tbl,
    transactions_tbl,
    user_request: Optional[str] = None,
) -> Dict[str, Any]:
    """
    Execute code in a controlled namespace.
    Accepts either raw Python code OR full content with
<execute_python> tags.
    Returns minimal artifacts: stdout, error, and extracted
answer.
    """
    # Extract code here (now centralized)
    code = _extract_execute_block(code_or_content)

    SAFE_GLOBALS = {
        "Query": Query,
```

```
        "get_current_balance": inv_utils.get_current_balance,
        "next_transaction_id": inv_utils.next_transaction_id,
        "user_request": user_request or "",
    }
    SAFE_LOCALS = {
        "db": db,
        "inventory_tbl": inventory_tbl,
        "transactions_tbl": transactions_tbl,
    }

    # Capture stdout from the executed code
    _stdout_buf, _old_stdout = io.StringIO(), sys.stdout
    sys.stdout = _stdout_buf
    err_text = None
    try:
        exec(code, SAFE_GLOBALS, SAFE_LOCALS)
    except Exception:
        err_text = traceback.format_exc()
    finally:
        sys.stdout = _old_stdout
    printed = _stdout_buf.getvalue().strip()

    # Extract possible answers set by the generated code
    answer = (
        SAFE_LOCALS.get("answer_text")
        or SAFE_LOCALS.get("answer_rows")
        or SAFE_LOCALS.get("answer_json")
    )


    return {
        "code": code,            # <- ya sin etiquetas
        "stdout": printed,
        "error": err_text,
        "answer": answer,
        "transactions_tbl": transactions_tbl.all(),  # For
inspection
        "inventory_tbl": inventory_tbl.all(),  # For inspection
    }

# Execute the generated plan for the round-sunglasses question
result = execute_generated_code(
    full_content_round,          # the full LLM response you
generated earlier
    db=db,
    inventory_tbl=inventory_tbl,
    transactions_tbl=transactions_tbl,
    user_request=prompt_round, # e.g., "Do you have any round
sunglasses in stock that are under $100?"
```

```
)

# Peek at exactly what Python the plan executed
utils.print_html(result["answer"], title="Plan Execution · Extracted Answer")
```

Plan Execution · Extracted Answer

```
Yes, we have our Classic round sunglasses for $60.
```

Putting it together

```
def customer_service_agent(
    question: str,
    *,
    db,
    inventory_tbl,
    transactions_tbl,
    model: str = "o4-mini",
    temperature: float = 1.0,
    reseed: bool = False,
) -> dict:
    """
    End-to-end helper:
      1) (Optional) reseed inventory & transactions
      2) Generate plan-as-code from `question`
      3) Execute in a controlled namespace
      4) Render before/after snapshots and return artifacts

    Returns:
      {
        "full_content": <raw LLM response (may include
<execute_python> tags)>,
        "exec": {
            "code": <extracted python>,
            "stdout": <plan logs>,
            "error": <traceback or None>,
            "answer": <answer_text/rows/json>,
            "inventory_after": [...],
            "transactions_after": [...]
        }
      }
    """
    # 0) Optional reseed
    if reseed:
        inv_utils.create_inventory()
        inv_utils.create_transactions()
```

```
    # 1) Show the question
    utils.print_html(question, title="User Question")

    # 2) Generate plan-as-code (FULL content)
    full_content = generate_llm_code(
        question,
        inventory_tbl=inventory_tbl,
        transactions_tbl=transactions_tbl,
        model=model,
        temperature=temperature,
    )
    utils.print_html(full_content, title="Plan with Code (Full
Response)")

    # 3) Before snapshots
    utils.print_html(json.dumps(inventory_tbl.all(), indent=2),
title="Inventory Table · Before")
    utils.print_html(json.dumps(transactions_tbl.all(),
indent=2), title="Transactions Table · Before")

    # 4) Execute
    exec_res = execute_generated_code(
        full_content,
        db=db,
        inventory_tbl=inventory_tbl,
        transactions_tbl=transactions_tbl,
        user_request=question,
    )

    # 5) After snapshots + final answer
    utils.print_html(exec_res["answer"], title="Plan Execution ·
Extracted Answer")
    utils.print_html(json.dumps(inventory_tbl.all(), indent=2),
title="Inventory Table · After")
    utils.print_html(json.dumps(transactions_tbl.all(),
indent=2), title="Transactions Table · After")

    # 6) Return artifacts
    return {
        "full_content": full_content,
        "exec": {
            "code": exec_res["code"],
            "stdout": exec_res["stdout"],
            "error": exec_res["error"],
            "answer": exec_res["answer"],
            "inventory_after": inventory_tbl.all(),
            "transactions_after": transactions_tbl.all(),
        },
    }
```

```
prompt = "I want to buy 3 pairs of classic sunglasses and 1 pair
of aviator sunglasses."

out = customer_service_agent(
    prompt,
    db=db,
    inventory_tbl=inventory_tbl,
    transactions_tbl=transactions_tbl,
    model="o4-mini",
    temperature=1.0,
    reseed=True,   # set False to keep current state of the
inventory and the transactions
)
```

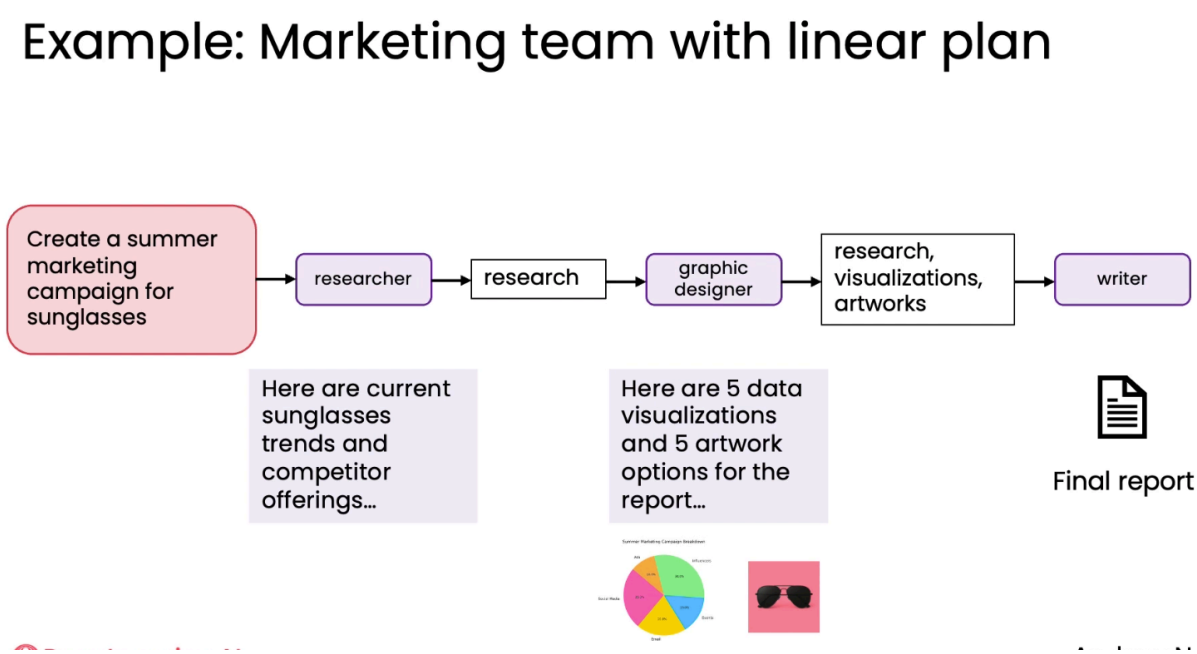
Multi-Agentic Workflows

Just like hiring a team with different roles, you can focus on building specialized agents for different subtasks rather than one agent trying to do everything.

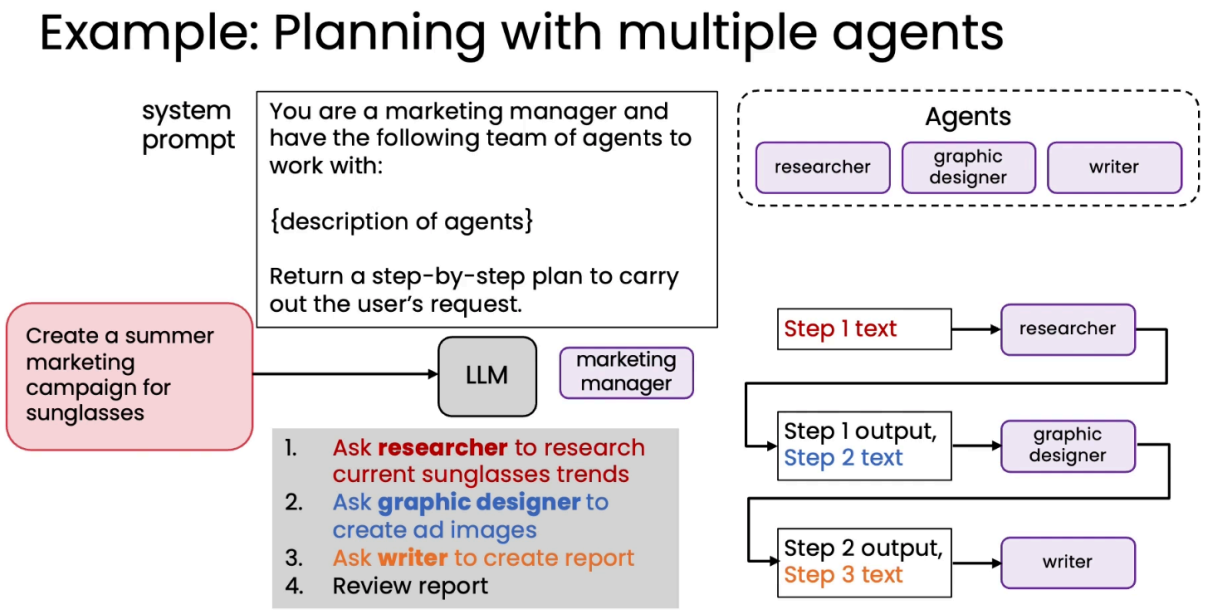
Some tasks require more than 1 person!

Task	Team
Create marketing assets	Researcher Graphic Designer Writer
Writing a research article	Researcher Statistician Lead writer Editor
Preparing a legal case	Associate Paralegal Investigator

Complex task can be decomposed into tasks that different people with different roles can perform.



The work on these agents can be done by different teams in parallel.



Project 2 - Market Research Team

The goal is to experience how multiple agents, tools, and models can be orchestrated into a single, coherent workflow.
learning how to combine the **imagination of large language models** with the **discipline of structured workflows**, giving you a practical pattern for building autonomous systems that are both creative and reliable.

The first tool is `tools.tavily_search_tool`, which performs live web searches to uncover evidence of current fashion trends. Try it now by running a simple query for *"trends in sunglasses fashion"*:

```
tools.tavily_search_tool('trends in sunglasses fashion')
```

The second tool is `tools.product_catalog_tool`, which returns the internal sunglasses catalog. Each entry includes details such as product name, ID, description, stock quantity, and price. This structured data will allow the agents to connect online fashion trends with actual items in stock:

```
tools.product_catalog_tool()
```

```
# ========================
# Imports
# ========================

# --- Standard library ---
import base64
import json
import os
import re
from datetime import datetime
from io import BytesIO

# --- Third-party ---
import requests
import openai
from PIL import Image
from dotenv import load_dotenv
from IPython.display import Markdown, display
import aisuite

# --- Local / project ---
import tools
import utils


# ========================
```

```
# Environment & Client
# ========================
load_dotenv()
client = aisuite.Client()


def market_research_agent(return_messages: bool = False):

    utils.log_agent_title_html("Market Research Agent", "🕵️")

    prompt_ = f"""
You are a fashion market research agent tasked with preparing a
trend analysis for a summer sunglasses campaign.

Your goal:
1. Explore current fashion trends related to sunglasses using web
search.
2. Review the internal product catalog to identify items that
align with those trends.
3. Recommend one or more products from the catalog that best
match emerging trends.
4. If needed, today date is {datetime.now().strftime("%Y-%m-
%d")}.

You can call the following tools:
- tavily_search_tool: to discover external web trends.
- product_catalog_tool: to inspect the internal sunglasses
catalog.

Once your analysis is complete, summarize:
- The top 2-3 trends you found.
- The product(s) from the catalog that fit these trends.
- A justification of why they are a good fit for the summer
campaign.
"""
    messages = [{"role": "user", "content": prompt_}]
    tools_ = tools.get_available_tools()

    while True:
        response = client.chat.completions.create(
            model="openai:o4-mini",
            messages=messages,
            tools=tools_,
            tool_choice="auto"
        )

        msg = response.choices[0].message

        if msg.content:
```

```
            utils.log_final_summary_html(msg.content)
            return (msg.content, messages) if return_messages
else msg.content

        if msg.tool_calls:
            for tool_call in msg.tool_calls:
                utils.log_tool_call_html(tool_call.function.name,
tool_call.function.arguments)
                result = tools.handle_tool_call(tool_call)
                utils.log_tool_result_html(result)

                messages.append(msg)

messages.append(tools.create_tool_response_message(tool_call,
result))
        else:
            utils.log_unexpected_html()
            return ("[⚠️ Unexpected: No tool_calls or content
returned]", messages) if return_messages else "[⚠️ Unexpected:
No tool_calls or content returned]"

market_research_result = market_research_agent()
def graphic_designer_agent(trend_insights: str, caption_style:
str = "short punchy", size: str = "1024x1024") -> dict:

    """
    Uses aisuite to generate a marketing prompt/caption and
OpenAI (directly) to generate the image.

    Args:
        trend_insights (str): Trend summary from the researcher
agent.
        caption_style (str): Optional style hint for caption.
        size (str): Image resolution (e.g., '1024x1024').

    Returns:
        dict: A dictionary with image_url, prompt, and caption.
    """


    utils.log_agent_title_html("Graphic Designer Agent", "🎨")

    # Step 1: Generate prompt and caption using aisuite
    system_message = (
        "You are a visual marketing assistant. Based on the input
trend insights, "
        "write a creative and visual prompt for an AI image
generation model, and also a short caption."
    )
```

```
    user_prompt = f"""
Trend insights:
{trend_insights}

Please output:
1. A vivid, descriptive prompt to guide image generation.
2. A marketing caption in style: {caption_style}.

Respond in this format:
{{"prompt": "...", "caption": "..."}}
"""

    chat_response = client.chat.completions.create(
        model="openai:o4-mini",
        messages=[
            {"role": "system", "content": system_message},
            {"role": "user", "content": user_prompt}
        ]
    )

    content = chat_response.choices[0].message.content.strip()
    match = re.search(r'\{.*\}', content, re.DOTALL)
    parsed = json.loads(match.group(0)) if match else {"error":
"No JSON returned", "raw": content}

    prompt = parsed["prompt"]
    caption = parsed["caption"]

    # Step 2: Generate image directly using openai-python
    openai_client = openai.OpenAI()

    image_response = openai_client.images.generate(
        model="dall-e-3",
        prompt=prompt,
        size=size,
        quality="standard",
        n=1,
        response_format="url"
    )

    image_url = image_response.data[0].url

    # Save image locally
    img_bytes = requests.get(image_url).content
    img = Image.open(BytesIO(img_bytes))

    filename = os.path.basename(image_url.split("?")[0])
    image_path = filename
```

```
    img.save(image_path)


    # Log summary with local image
    utils.log_final_summary_html(f"""
        <h3>Generated Image and Caption</h3>

        <p><strong>Image Path:</strong> <code>{image_path}</
code></p>

        <p><strong>Generated Image:</strong></p>
        <img src="{image_path}" alt="Generated Image" style="max-
width: 100%; height: auto; border: 1px solid #ccc; border-radius:
8px; margin-top: 10px; margin-bottom: 10px;">

        <p><strong>Prompt:</strong> {prompt}</p>
    """)


    return {
        "image_url": image_url,
        "prompt": prompt,
        "caption": caption,
        "image_path": image_path
    }

graphic_designer_agent_result = graphic_designer_agent(
    trend_insights=market_research_result,
)
def copywriter_agent(image_path: str, trend_summary: str, model:
str = "openai:o4-mini") -> dict:

    """
    Uses aisuite (OpenAI only) to send an image and a trend
summary and return a campaign quote.

    Args:
        image_path (str): URL of the image to be analyzed.
        trend_summary (str): Text from the researcher agent.
        model (str): OpenAI model (e.g., openai:o4-mini,
openai:gpt-4o)

    Returns:
        dict: {
            "quote": "...",
            "justification": "...",
            "image_path": "..."
        }
    """
```

```
    utils.log_agent_title_html("Copywriter Agent", "✍️")

    # Step 1: Load local image and encode as base64
    with open(image_path, "rb") as f:
        img_bytes = f.read()

    b64_img = base64.b64encode(img_bytes).decode("utf-8")

    # Step 2: Build OpenAI-compliant multimodal message
    messages = [
        {
            "role": "system",
            "content": "You are a copywriter that creates elegant
campaign quotes based on an image and a marketing trend summary."
        },
        {
            "role": "user",
            "content": [
                {
                    "type": "image_url",
                    "image_url": {
                        "url": f"
{b64_img}",
                        "detail": "auto"
                    }
                },
                {
                    "type": "text",
                    "text": f"""
Here is a visual marketing image and a trend analysis:

Trend summary:
\"\"\"{trend_summary}\"\"\"

Please return a JSON object like:
{{
  "quote": "A short, elegant campaign phrase (max 12 words)",
  "justification": "Why this quote matches the image and trend"
}}"""
                }
            ]
        }
    ]

    # Step 3: Send request via aisuite
    response = client.chat.completions.create(
        model=model,
        messages=messages,
```

```
    )

    # Step 4: Parse JSON response
    content = response.choices[0].message.content.strip()

    utils.log_final_summary_html(content)

    try:
        match = re.search(r'\{.*\}', content, re.DOTALL)
        parsed = json.loads(match.group(0)) if match else
{"error": "No valid JSON returned"}
    except Exception as e:
        parsed = {"error": f"Failed to parse: {e}", "raw":
content}


    parsed["image_path"] = image_path
    return parsed

copywriter_agent_result = copywriter_agent(
    image_path=graphic_designer_agent_result["image_path"],
    trend_summary=market_research_result,
)
def packaging_agent(trend_summary: str, image_url: str, quote:
str, justification: str, output_path: str =
"campaign_summary.md") -> str:

    """
    Packages the campaign assets into a beautifully formatted
markdown report for executive review.

    Args:
        trend_summary (str): Summary of the market trends.
        image_url (str): URL of the campaign image.
        quote (str): Marketing quote to overlay.
        justification (str): Explanation for the quote.
        output_path (str): Path to save the markdown report.

    Returns:
        str: Path to the saved markdown file.
    """


    utils.log_agent_title_html("Packaging Agent", "📦")

    # We use this path in the src of the <img>
    styled_image_html = f"""
![Open the generated file to see]({image_url})
    """
```

```
    beautified_summary = client.chat.completions.create(
        model="openai:o4-mini",
        messages=[
            {"role": "system", "content": "You are a marketing
communication expert writing elegant campaign summaries for
executives."},
            {"role": "user", "content": f"""
Please rewrite the following trend summary to be clear,
professional, and engaging for a CEO audience:

\"\"\"{trend_summary.strip()}\"\"\"
"""}
        ]
    ).choices[0].message.content.strip()

    utils.log_tool_result_html(beautified_summary)

    # Combine all parts into markdown
    markdown_content = f"""# 🕶️ Summer Sunglasses Campaign —
Executive Summary

## 📊 Refined Trend Insights
{beautified_summary}

## 🎯 Campaign Visual
{styled_image_html}

## ✍️ Campaign Quote
{quote.strip()}

## ✅ Why This Works
{justification.strip()}

---

*Report generated on {datetime.now().strftime('%Y-%m-%d')}*
"""

    with open(output_path, "w", encoding="utf-8") as f:
        f.write(markdown_content)

    return output_path
packaging_agent_result = packaging_agent(
    trend_summary=market_research_result,
    image_url=graphic_designer_agent_result["image_path"],
    quote=copywriter_agent_result["quote"],
    justification=copywriter_agent_result["justification"],
```

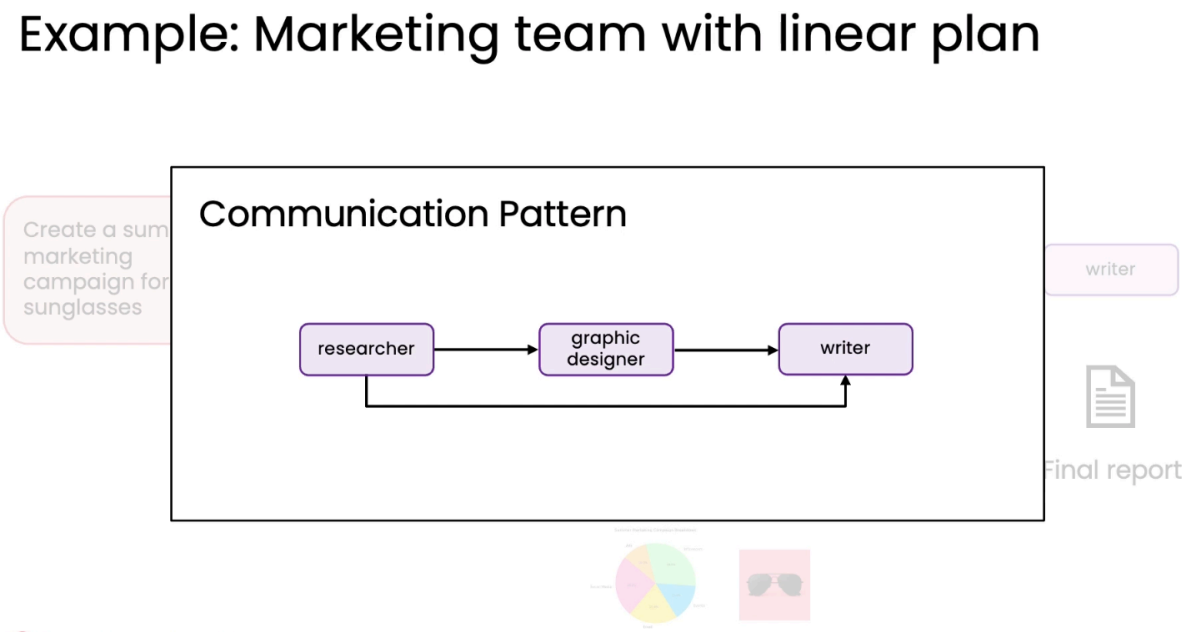
```
    output_path=f"campaign_summary_{datetime.now().strftime('%Y-
%m-%d_%H-%M-%S')}.md"
)
# Load and render the markdown content
with open(packaging_agent_result, "r", encoding="utf-8") as f:
    md_content = f.read()

display(Markdown(md_content))
def run_sunglasses_campaign_pipeline(output_path: str =
"campaign_summary.md") -> dict:
    """
    Runs the full summer sunglasses campaign pipeline:
    1. Market research (search trends + match products)
    2. Generate visual + caption
    3. Generate quote based on image + trend
    4. Create executive markdown report

    Returns:
        dict: Dictionary containing all intermediate results +
path to final report
    """
    # 1. Run market research agent
    trend_summary = market_research_agent()
    print("✅ Market research completed")

    # 2. Generate image + caption
    visual_result =
graphic_designer_agent(trend_insights=trend_summary)
    image_path = visual_result["image_path"]
    print("🖼️ Image generated")

    # 3. Generate quote based on image + trends
    quote_result = copywriter_agent(image_path=image_path,
trend_summary=trend_summary)
    quote = quote_result.get("quote", "")
    justification = quote_result.get("justification", "")
    print("💬 Quote created")

    # 4. Generate markdown report
    md_path = packaging_agent(
        trend_summary=trend_summary,
        image_url=image_path,
        quote=quote,
        justification=justification,

output_path=f"campaign_summary_{datetime.now().strftime('%Y-%m-
%d_%H-%M-%S')}.md"
    )
```

```
    print(f"📦 Report generated: {md_path}")

    return {
        "trend_summary": trend_summary,
        "visual": visual_result,
        "quote": quote_result,
        "markdown_path": md_path
    }

results = run_sunglasses_campaign_pipeline()
with open(results["markdown_path"], "r", encoding="utf-8") as f:
    md_content = f.read()
display(Markdown(md_content))
```

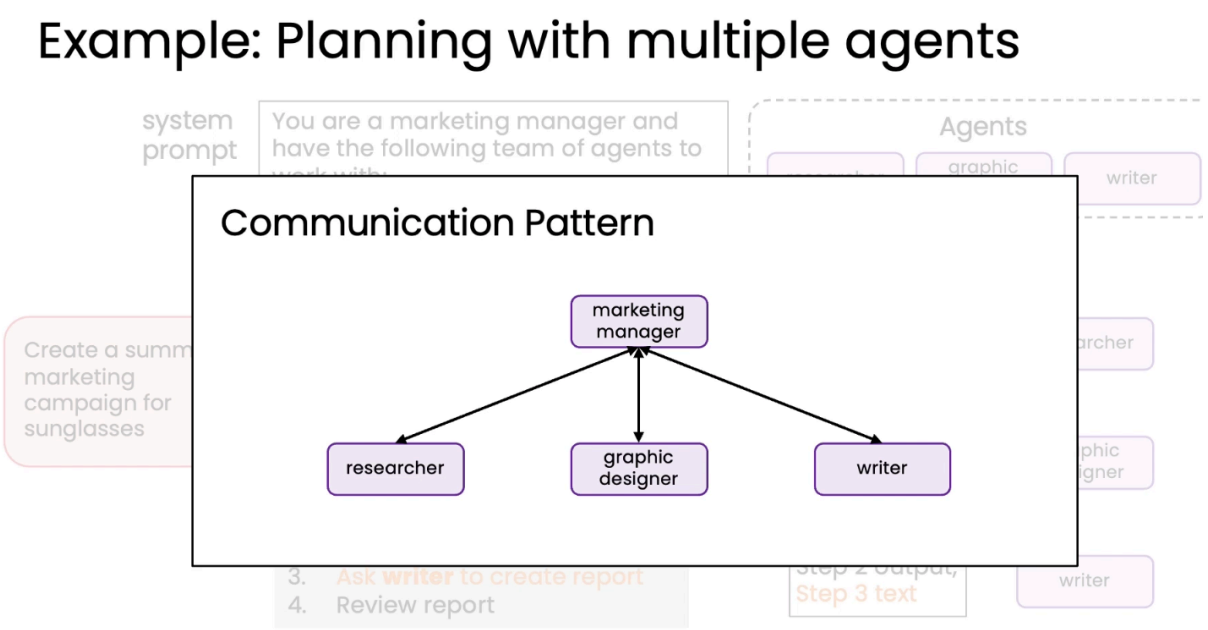
Communication patterns for multi-agent systems

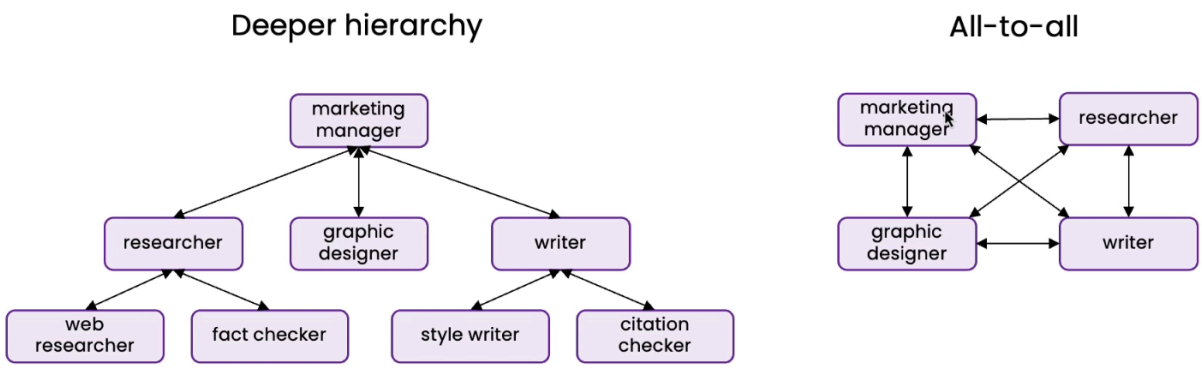
Linear patterns pass outputs stepwise between agents; hierarchical patterns use a manager to coordinate; all-to-all allows unrestricted agent communication.

Linear pattern



Hierarchical pattern





Project 3 - Multi-agent workflow

- **Planning Agent / Writer**: Creates an outline and coordinates tasks.
- **Research Agent**: Gathers external information using tools like Arxiv, Tavily, and Wikipedia.
- **Editor Agent**: Reflects on the report and provides suggestions for improvement.

```
# ========================
# Imports
```

```
# ========================

# --- Standard library
from datetime import datetime
import re
import json
import ast


# --- Third-party ---
from IPython.display import Markdown, display
from aisuite import Client

# --- Local / project ---
import research_tools
CLIENT = Client()
def planner_agent(topic: str, model: str = "openai:o4-mini") ->
list[str]:
    """
    Generates a plan as a Python list of steps (strings) for a
research workflow.

    Args:
        topic (str): Research topic to investigate.
        model (str): Language model to use.

    Returns:
        List[str]: A list of executable step strings.
    """


    # Build the user prompt
    user_prompt = f"""
    You are a planning agent responsible for organizing a
research workflow with multiple intelligent agents.

    🧠 Available agents:
    - A research agent who can search the web, Wikipedia, and
arXiv.
    - A writer agent who can draft research summaries.
    - An editor agent who can reflect and revise the drafts.

    🎯 Your job is to write a clear, step-by-step research plan
**as a valid Python list**, where each step is a string.
    Each step should be atomic, executable, and must rely only on
the capabilities of the above agents.

    🚫 DO NOT include irrelevant tasks like "create CSV", "set
```

```
up a repo", "install packages", etc.
    ✅ DO include real research-related tasks (e.g., search,
summarize, draft, revise).
    ✅ DO assume tool use is available.
    ✅ DO NOT include explanation text — return ONLY the Python
list.
    ✅ The final step should be to generate a Markdown document
containing the complete research report.

    Topic: "{topic}"
    """

    # Add the user prompt to the messages list
    messages = [{"role": "user", "content": user_prompt}]

    ### START CODE HERE ###

    # Call the LLM
    response = CLIENT.chat.completions.create(
        # Pass in the model
        model=model,
        # Define the messages. Remember this is meant to be a
user prompt!
        messages=messages,
        # Keep responses creative
        temperature=1,
    )

    ### END CODE HERE ###

    # Extract message from response
    steps_str = response.choices[0].message.content.strip()

    # Parse steps
    steps = ast.literal_eval(steps_str)

    return steps

def research_agent(task: str, model: str = "openai:gpt-4o",
return_messages: bool = False):
    """
    Executes a research task using tools via aisuite (no manual
loop).
    Returns either the assistant text, or (text, messages) if
return_messages=True.
    """
    print("==================================")
```

```
    print("🔍 Research Agent")
    print("================================")

    current_time = datetime.now().strftime('%Y-%m-%d')

    ### START CODE HERE ###
    role = "research assistant"

    # Create a customizable prompt by defining the role (e.g.,
"research assistant"),
    # listing tools (arxiv_tool, tavily_tool, wikipedia_tool) for
various searches,
    # specifying the task with a placeholder, and including a
current_time placeholder.
    prompt = f"""You are acting as a {role}.

You have access to the following tools for searching and
gathering information:

arxiv_tool (for academic papers and preprints)

tavily_tool (for web and general online searches)

wikipedia_tool (for concise knowledge and background)

Your task: {task}

Reference the current time as needed: {current_time}

Please use the available tools appropriately to fulfill your
task, clearly organizing your findings and presenting results in
a structured format.

    """

    # Create the messages dict to pass to the LLM. Remember this
is a user prompt!
    messages = [{"role": "user", "content": prompt}]

    # Save all of your available tools in the tools list. These
can be found in the research_tools module.
    # You can identify each tool in your list like this:
    # research_tools.<name_of_tool>, where <name_of_tool> is
replaced with the function name of the tool.
    tools = []

    # Call the model with tools enabled
    response = CLIENT.chat.completions.create(
        # Set the model
```

```
        model=model,
        # Pass in the messages. You already defined this!
        messages=messages,
        # Pass in the tools list. You already defined this!
        tools=[research_tools.arxiv_search_tool,
research_tools.tavily_search_tool,
research_tools.wikipedia_search_tool],
        # Set the LLM to automatically choose the tools
        tool_choice="auto",
        # Set the max turns to 6
        max_turns=6
    )

    ### END CODE HERE ###

    content = response.choices[0].message.content
    print("✅ Output:\n", content)


    return (content, messages) if return_messages else content

def writer_agent(task: str, model: str = "openai:gpt-4o") -> str:
# @REPLACE def writer_agent(task: str, model: str = None) -> str:
    """
    Executes writing tasks, such as drafting, expanding, or
summarizing text.
    """
    print("==================================")
    print("✍️ Writer Agent")
    print("==================================")

    ### START CODE HERE ###

    # Create the system prompt.
    # This should assign the LLM the role of a writing agent
specialized in generating well-structured academic or technical
content
    system_prompt = f"""You are an expert writing agent
specializing in generating, expanding, and summarizing academic
and technical content.
Your tasks may involve:

Creating first drafts from simple prompts

Expanding outlines or bullet points into well-developed academic
prose

Summarizing complex documents clearly and concisely
```

```
Ensuring logical structure, coherence, and adherence to academic
or technical conventions in all outputs

Using a formal, precise, and accessible tone suited for research,
reports, or publication

Always follow instructions carefully, present material in well-
organized paragraphs, and focus on clarity and rigor.
    """

    # Define the system msg by using the system_prompt and
assigning the role of system
    system_msg = {"role": "system", "content": system_prompt}

    # Define the user msg. In this case the user prompt should be
the task passed to the function
    user_msg = {"role": "user", "content": task}

    # Add both system and user messages to the messages list
    messages = [system_msg, user_msg]

    ### END CODE HERE ###

    response = CLIENT.chat.completions.create(
        model=model,
        messages=messages,
        temperature=1.0
    )

    return response.choices[0].message.content

def editor_agent(task: str, model: str = "openai:gpt-4o") -> str:
    """
    Executes editorial tasks such as reflection, critique, or
revision.
    """
    print("==================================")
    print("🧠 Editor Agent")
    print("==================================")

    ### START CODE HERE ###

    # Create the system prompt.
    # This should assign the LLM the role of an editor agent
specialized in reflecting on, critiquing, or improving existing
drafts.
    system_prompt = f"""You are an expert editor agent
specializing in academic and technical writing.
Your responsibilities include:
```

```
Reflecting critically on written drafts, identifying both
strengths and limitations

Providing constructive feedback and actionable suggestions for
refinement

Rewriting or improving drafts for clarity, organization,
coherence, and adherence to academic or technical standards

Ensuring that revisions enhance argumentation, depth of analysis,
style, and readability

Apply rigorous editorial judgment to every task. Always
communicate feedback clearly and revise content for maximum
effectiveness and professionalism.
    """

    # Define the system msg by using the system_prompt and
assigning the role of system
    system_msg = {"role": "system", "content": system_prompt}

    # Define the user msg. In this case the user prompt should be
the task passed to the function
    user_msg = {"role": "user", "content": task}

    # Add both system and user messages to the messages list
    messages = [system_msg, user_msg]

    ### END CODE HERE ###

    response = CLIENT.chat.completions.create(
        model=model,
        messages=messages,
        temperature=0.7
    )

    return response.choices[0].message.content

agent_registry = {
    "research_agent": research_agent,
    "editor_agent": editor_agent,
    "writer_agent": writer_agent,
}

def clean_json_block(raw: str) -> str:
    """
    Clean the contents of a JSON block that may come wrapped with
Markdown backticks.
```

```
    """
    raw = raw.strip()
    if raw.startswith("```"):
        raw = re.sub(r"^```(?:json)?\n?", "", raw)
        raw = re.sub(r"\n?```$", "", raw)
    return raw.strip()

def executor_agent(topic, model: str = "openai:gpt-4o",
limit_steps: bool = True):

    plan_steps = planner_agent(topic)
    max_steps = 4

    if limit_steps:
        plan_steps = plan_steps[:min(len(plan_steps), max_steps)]

    history = []

    print("==================================")
    print("🎯 Editor Agent")
    print("==================================")

    for i, step in enumerate(plan_steps):

        agent_decision_prompt = f"""
        You are an execution manager for a multi-agent research
team.

        Given the following instruction, identify which agent
should perform it and extract the clean task.

        Return only a valid JSON object with two keys:
        - "agent": one of ["research_agent", "editor_agent",
"writer_agent"]
        - "task": a string with the instruction that the agent
should follow

        Only respond with a valid JSON object. Do not include
explanations or markdown formatting.

        Instruction: "{step}"
        """
        response = CLIENT.chat.completions.create(
            model=model,
            messages=[{"role": "user", "content":
agent_decision_prompt}],
            temperature=0,
        )
```

```
            raw_content = response.choices[0].message.content
            cleaned_json = clean_json_block(raw_content)
            agent_info = json.loads(cleaned_json)

            agent_name = agent_info["agent"]
            task = agent_info["task"]

            context = "\n".join([
                f"Step {j+1} executed by {a}:\n{r}"
                for j, (s, a, r) in enumerate(history)
            ])
            enriched_task = f"""
            You are {agent_name}.

            Here is the context of what has been done so far:
            {context}

            Your next task is:
            {task}
            """

            print(f"\n🛠️ Executing with agent: `{agent_name}` on
task: {task}")

            if agent_name in agent_registry:
                output = agent_registry[agent_name](enriched_task)
                history.append((step, agent_name, output))
            else:
                output = f"⚠️ Unknown agent: {agent_name}"
                history.append((step, agent_name, output))

            print(f"✅ Output:\n{output}")

        return history

    # If you want to see the full workflow without limiting the
    number of steps. Set limit_steps to False
    # Keep in mind this could take more than 10 minutes to complete
    executor_history = executor_agent("The ensemble Kalman filter for
    time series forecasting", limit_steps=True)

    md = executor_history[-1][-1].strip("`")
    display(Markdown(md))
```

The output:

```
==================================
🎯 Editor Agent
==================================

🛠️ Executing with agent: `research_agent` on task:
Perform a comprehensive literature search for 'ensemble
Kalman filter' and 'time series forecasting' on arXiv,
the web, and Wikipedia.
==================================
🔍 Research Agent
==================================
✅ Output:
 **Comprehensive Literature Search on 'Ensemble Kalman
Filter' and 'Time Series Forecasting'**

---

### Ensemble Kalman Filter

#### ArXiv Papers

1. **"An Explicit Probabilistic Derivation of Inflation
in a Scalar Ensemble Kalman Filter"**
   - *Authors:* Andrey A Popov, Adrian Sandu
   - *Summary:* Analyzes convergence of an ensemble
Kalman filter solution to an exact filter solution in
scalar cases, emphasizing probabilistic approaches.
   - [Read more](http://arxiv.org/abs/2003.13162v1)

2. **"Derivation of Ensemble Kalman-Bucy Filters with
Unbounded Nonlinear Coefficients"**
   - *Author:* Theresa Lange
   - *Summary:* Discusses the continuous-time limits of
the Ensemble Kalman Filter with nonlinear models.
   - [Read more](http://arxiv.org/abs/2012.07572v3)
```

```
3. **"Convergence of the Square Root Ensemble Kalman
Filter in the Large Ensemble Limit"**
   - *Authors:* Evan Kwiatkowski, Jan Mandel
   - *Summary:* Examines convergence properties of an
unbiased square root ensemble filter using
deterministic algorithms.
   - [Read more](http://arxiv.org/abs/1404.4093v2)

4. **"Particle Kalman Filtering: A Nonlinear Bayesian
Framework for Ensemble Kalman Filters"**
   - *Authors:* Ibrahim Hoteit, Xiaodong Luo, Dinh-Tuan
Pham
   - *Summary:* Proposes the particle Kalman filter, an
evolution of the traditional Kalman and particle
filters.
   - [Read more](http://arxiv.org/abs/1108.0168v1)

5. **"On the Continuous Time Limit of the Ensemble
Kalman Filter"**
   - *Authors:* Theresa Lange, Wilhelm Stannat
   - *Summary:* Explores continuous time limits and
interactions in Ensemble Kalman Filter algorithms.
   - [Read more](http://arxiv.org/abs/1901.05204v1)

#### Wikipedia Summary
- **Ensemble Kalman Filter (EnKF):** A Monte Carlo
implementation of the Bayesian update problem applied
to large-scale systems. It is widely used in
geophysical sciences for data assimilation and ensemble
forecasting. [Learn more](https://en.wikipedia.org/
wiki/Ensemble_Kalman_filter)

#### Tavily Web Search Highlights
- Discusses various implementations and practical
applications of the Ensemble Kalman Filter, emphasizing
its efficiency in large-scale systems and its relation
to particle filters.
- [Explore further](https://asp-
```

```
eurasipjournals.springeropen.com/articles/10.1186/
s13634-017-0492-x)

---

### Time Series Forecasting

#### ArXiv Papers

1. **"Inter Time Series Sales Forecasting"**
   - *Author:* Manisha Gahirwal
   - *Summary:* Explores the combination of different
forecasting models for improved sales predictions.
   - [Read more](http://arxiv.org/abs/1303.0117v1)

2. **"Optimizing Time Series Forecasting Architectures:
A Hierarchical Neural Architecture Search Approach"**
   - *Authors:* Difan Deng, Marius Lindauer
   - *Summary:* Introduces a hierarchical search to
optimize deep learning architectures for time series
forecasting.
   - [Read more](http://arxiv.org/abs/2406.05088v1)

3. **"Monash Time Series Forecasting Archive"**
   - *Authors:* Rakshitha Godahewa et al.
   - *Summary:* Presents a comprehensive archive of
diverse time series datasets for research.
   - [Read more](http://arxiv.org/abs/2105.06643v1)

4. **"For2For: Learning to Forecast from Forecasts"**
   - *Authors:* Shi Zhao, Ying Feng
   - *Summary:* Combines traditional forecasting
methods with ML models for improved accuracy.
   - [Read more](http://arxiv.org/abs/2001.04601v1)

5. **"TimeGym: Debugging for Time Series Modeling in
Python"**
   - *Author:* Diogo Seca
```

```
   - *Summary:* Offers a toolkit for debugging
forecasting models, focusing on pipeline testing.
   - [Read more](http://arxiv.org/abs/2105.01404v1)

#### Wikipedia Summary
- **Time Series Forecasting:** Uses historical data to
predict future values, employing models like ARIMA,
machine learning, and deep learning techniques for
strategic decisions. [Learn more](https://
en.wikipedia.org/wiki/Time_series)

#### Tavily Web Search Highlights
- Highlights the application of time series data in
various fields such as economics, healthcare, and
technology, focusing on predictive modeling and
optimization strategies.
- [Explore further](https://www.tableau.com/analytics/
time-series-forecasting)

This comprehensive literature search provides a robust
foundation for exploring the application and
development of Ensemble Kalman Filters and Time Series
Forecasting in contemporary research and practical
applications.
✅ Output:
**Comprehensive Literature Search on 'Ensemble Kalman
Filter' and 'Time Series Forecasting'**

---

### Ensemble Kalman Filter

#### ArXiv Papers

1. **"An Explicit Probabilistic Derivation of Inflation
in a Scalar Ensemble Kalman Filter"**
   - *Authors:* Andrey A Popov, Adrian Sandu
```

```
   - *Summary:* Analyzes convergence of an ensemble
Kalman filter solution to an exact filter solution in
scalar cases, emphasizing probabilistic approaches.
   - [Read more](http://arxiv.org/abs/2003.13162v1)

2. **"Derivation of Ensemble Kalman-Bucy Filters with
Unbounded Nonlinear Coefficients"**
   - *Author:* Theresa Lange
   - *Summary:* Discusses the continuous-time limits of
the Ensemble Kalman Filter with nonlinear models.
   - [Read more](http://arxiv.org/abs/2012.07572v3)

3. **"Convergence of the Square Root Ensemble Kalman
Filter in the Large Ensemble Limit"**
   - *Authors:* Evan Kwiatkowski, Jan Mandel
   - *Summary:* Examines convergence properties of an
unbiased square root ensemble filter using
deterministic algorithms.
   - [Read more](http://arxiv.org/abs/1404.4093v2)

4. **"Particle Kalman Filtering: A Nonlinear Bayesian
Framework for Ensemble Kalman Filters"**
   - *Authors:* Ibrahim Hoteit, Xiaodong Luo, Dinh-Tuan
Pham
   - *Summary:* Proposes the particle Kalman filter, an
evolution of the traditional Kalman and particle
filters.
   - [Read more](http://arxiv.org/abs/1108.0168v1)

5. **"On the Continuous Time Limit of the Ensemble
Kalman Filter"**
   - *Authors:* Theresa Lange, Wilhelm Stannat
   - *Summary:* Explores continuous time limits and
interactions in Ensemble Kalman Filter algorithms.
   - [Read more](http://arxiv.org/abs/1901.05204v1)

#### Wikipedia Summary
- **Ensemble Kalman Filter (EnKF):** A Monte Carlo
```

```
implementation of the Bayesian update problem applied
to large-scale systems. It is widely used in
geophysical sciences for data assimilation and ensemble
forecasting. [Learn more](https://en.wikipedia.org/
wiki/Ensemble_Kalman_filter)

#### Tavily Web Search Highlights
- Discusses various implementations and practical
applications of the Ensemble Kalman Filter, emphasizing
its efficiency in large-scale systems and its relation
to particle filters.
- [Explore further](https://asp-
eurasipjournals.springeropen.com/articles/10.1186/
s13634-017-0492-x)

---

### Time Series Forecasting

#### ArXiv Papers

1. **"Inter Time Series Sales Forecasting"**
   - *Author:* Manisha Gahirwal
   - *Summary:* Explores the combination of different
forecasting models for improved sales predictions.
   - [Read more](http://arxiv.org/abs/1303.0117v1)

2. **"Optimizing Time Series Forecasting Architectures:
A Hierarchical Neural Architecture Search Approach"**
   - *Authors:* Difan Deng, Marius Lindauer
   - *Summary:* Introduces a hierarchical search to
optimize deep learning architectures for time series
forecasting.
   - [Read more](http://arxiv.org/abs/2406.05088v1)

3. **"Monash Time Series Forecasting Archive"**
   - *Authors:* Rakshitha Godahewa et al.
   - *Summary:* Presents a comprehensive archive of
```

```
diverse time series datasets for research.
   - [Read more](http://arxiv.org/abs/2105.06643v1)

4. **"For2For: Learning to Forecast from Forecasts"**
   - *Authors:* Shi Zhao, Ying Feng
   - *Summary:* Combines traditional forecasting
methods with ML models for improved accuracy.
   - [Read more](http://arxiv.org/abs/2001.04601v1)

5. **"TimeGym: Debugging for Time Series Modeling in
Python"**
   - *Author:* Diogo Seca
   - *Summary:* Offers a toolkit for debugging
forecasting models, focusing on pipeline testing.
   - [Read more](http://arxiv.org/abs/2105.01404v1)

#### Wikipedia Summary
- **Time Series Forecasting:** Uses historical data to
predict future values, employing models like ARIMA,
machine learning, and deep learning techniques for
strategic decisions. [Learn more](https://
en.wikipedia.org/wiki/Time_series)

#### Tavily Web Search Highlights
- Highlights the application of time series data in
various fields such as economics, healthcare, and
technology, focusing on predictive modeling and
optimization strategies.
- [Explore further](https://www.tableau.com/analytics/
time-series-forecasting)

This comprehensive literature search provides a robust
foundation for exploring the application and
development of Ensemble Kalman Filters and Time Series
Forecasting in contemporary research and practical
applications.
🛠️ Executing with agent: `research_agent` on task:
```

```
Identify and collect key publications, definitions, and
core equations of the ensemble Kalman filter relevant
to time series forecasting.
==================================
🔍 Research Agent
==================================
✅ Output:
 ### Key Publications and Information on Ensemble
Kalman Filter Relevant to Time Series Forecasting

#### **ArXiv Publications**

1. **"On-Line Learning of Linear Dynamical Systems:
Exponential Forgetting in Kalman Filters"**
   - *Authors:* Mark Kozdoba, Jakub Marecek, Tigran
Tchrakian, Shie Mannor
   - *Summary:* Examines how the Kalman filter's
prediction dependence on past data decays exponentially
with non-degenerate process noise. This study explores
using only recent observations for forecasting,
providing a practical online algorithm for learning
linear dynamical systems.
   - [Read more](http://arxiv.org/abs/1809.05870v1) |
[PDF](http://arxiv.org/pdf/1809.05870v1)

2. **"Long-time accuracy of ensemble Kalman filters for
chaotic and machine-learned dynamical systems"**
   - *Authors:* Daniel Sanz-Alonso, Nathan Waniorek
   - *Summary:* Establishes conditions under which
ensemble Kalman filters maintain small estimation
errors in the long-term for partially-observed chaotic
dynamical systems. The study validates combining
ensemble Kalman filters with machine-learned models for
data assimilation.
   - [Read more](http://arxiv.org/abs/2412.14318v1) |
[PDF](http://arxiv.org/pdf/2412.14318v1)
```

```
3. **"Ensemble Kalman Filter with perturbed
observations in weather forecasting and data
assimilation"**
   - *Author:* Yihua Yang
   - *Summary:* Engages with data assimilation using
Ensemble Kalman Filters in weather forecasting. Focuses
on improving efficiency by considering perturbed
observations, employing Monte Carlo simulations for
non-linear systems.
   - [Read more](http://arxiv.org/abs/2004.04275v2) |
[PDF](http://arxiv.org/pdf/2004.04275v2)

4. **"Generating Trading Signals by ML algorithms or
time series ones?"**
   - *Author:* Omid Safarzadeh
   - *Summary:* Investigates trading signal generation
using Kalman Filter and machine learning methods like
Random Forests, highlighting comparative advantages.
   - [Read more](http://arxiv.org/abs/2007.11098v1) |
[PDF](http://arxiv.org/pdf/2007.11098v1)

5. **"Effects of Observational Data Shortage on
Accuracy of Global Solar Activity Forecast"**
   - *Author:* Irina N. Kitiashvili
   - *Summary:* Discusses the use of the Ensemble
Kalman Filter for solar activity forecasting,
particularly under constraints of short observational
data series.
   - [Read more](http://arxiv.org/abs/2001.09376v2) |
[PDF](http://arxiv.org/pdf/2001.09376v2)

#### **Wikipedia Summary: Ensemble Kalman Filter
Equations**

The **Ensemble Kalman Filter (EnKF)** is a recursive
filter appropriate for complex systems with numerous
variables, particularly in geophysical models. It
functions as a Monte Carlo implementation of the
```

```
Bayesian update problem, utilizing the ensemble to
approximate the covariance matrix. The EnKF assumes
Gaussian distributions and optimizes data assimilation
in ensemble forecasting compared to particle filtering.

- More information can be found on this Wikipedia
[page](https://en.wikipedia.org/wiki/
Ensemble_Kalman_filter).
✅ Output:
### Key Publications and Information on Ensemble Kalman
Filter Relevant to Time Series Forecasting

#### **ArXiv Publications**

1. **"On-Line Learning of Linear Dynamical Systems:
Exponential Forgetting in Kalman Filters"**
   - *Authors:* Mark Kozdoba, Jakub Marecek, Tigran
Tchrakian, Shie Mannor
   - *Summary:* Examines how the Kalman filter's
prediction dependence on past data decays exponentially
with non-degenerate process noise. This study explores
using only recent observations for forecasting,
providing a practical online algorithm for learning
linear dynamical systems.
   - [Read more](http://arxiv.org/abs/1809.05870v1) |
[PDF](http://arxiv.org/pdf/1809.05870v1)

2. **"Long-time accuracy of ensemble Kalman filters for
chaotic and machine-learned dynamical systems"**
   - *Authors:* Daniel Sanz-Alonso, Nathan Waniorek
   - *Summary:* Establishes conditions under which
ensemble Kalman filters maintain small estimation
errors in the long-term for partially-observed chaotic
dynamical systems. The study validates combining
ensemble Kalman filters with machine-learned models for
data assimilation.
   - [Read more](http://arxiv.org/abs/2412.14318v1) |
```

```
[PDF](http://arxiv.org/pdf/2412.14318v1)

3. **"Ensemble Kalman Filter with perturbed
observations in weather forecasting and data
assimilation"**
   - *Author:* Yihua Yang
   - *Summary:* Engages with data assimilation using
Ensemble Kalman Filters in weather forecasting. Focuses
on improving efficiency by considering perturbed
observations, employing Monte Carlo simulations for
non-linear systems.
   - [Read more](http://arxiv.org/abs/2004.04275v2) |
[PDF](http://arxiv.org/pdf/2004.04275v2)

4. **"Generating Trading Signals by ML algorithms or
time series ones?"**
   - *Author:* Omid Safarzadeh
   - *Summary:* Investigates trading signal generation
using Kalman Filter and machine learning methods like
Random Forests, highlighting comparative advantages.
   - [Read more](http://arxiv.org/abs/2007.11098v1) |
[PDF](http://arxiv.org/pdf/2007.11098v1)

5. **"Effects of Observational Data Shortage on
Accuracy of Global Solar Activity Forecast"**
   - *Author:* Irina N. Kitiashvili
   - *Summary:* Discusses the use of the Ensemble
Kalman Filter for solar activity forecasting,
particularly under constraints of short observational
data series.
   - [Read more](http://arxiv.org/abs/2001.09376v2) |
[PDF](http://arxiv.org/pdf/2001.09376v2)

#### **Wikipedia Summary: Ensemble Kalman Filter
Equations**

The **Ensemble Kalman Filter (EnKF)** is a recursive
filter appropriate for complex systems with numerous
```

```
variables, particularly in geophysical models. It
functions as a Monte Carlo implementation of the
Bayesian update problem, utilizing the ensemble to
approximate the covariance matrix. The EnKF assumes
Gaussian distributions and optimizes data assimilation
in ensemble forecasting compared to particle filtering.

- More information can be found on this Wikipedia
[page](https://en.wikipedia.org/wiki/
Ensemble_Kalman_filter).

🛠️ Executing with agent: `research_agent` on task:
Summarize state-of-the-art modifications and
applications of the ensemble Kalman filter in
forecasting contexts.
==================================
🔍 Research Agent
==================================
```