Executable Code Actions Elicit Better LLM Agents

Xingyao Wang [1] Yangyi Chen [1] Lifan Yuan [1] Yizhe Zhang [2] Hao Peng [1] Heng Ji [1]

4 Jun 2024

Presented by:

John Tan Chong Min

Code action space has a higher success rate

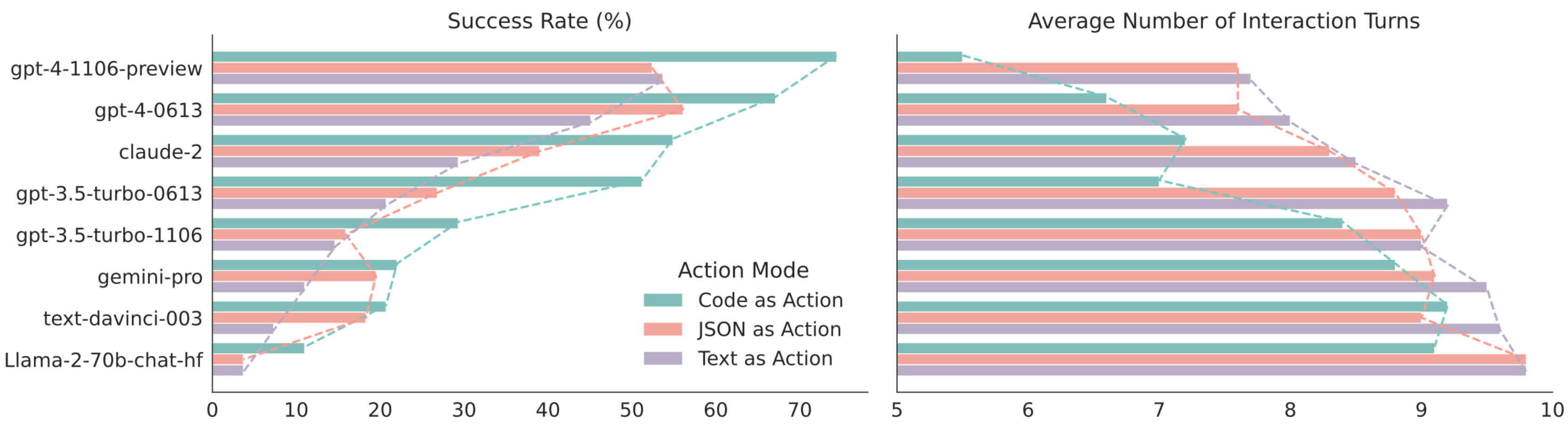


Figure 1: Comparison between `CodeAct` and Text / JSON as action. **(top)** Illustrative example comparing different actions. **(bottom)** Quantitative results on M^{3}ToolEval (§2.3).

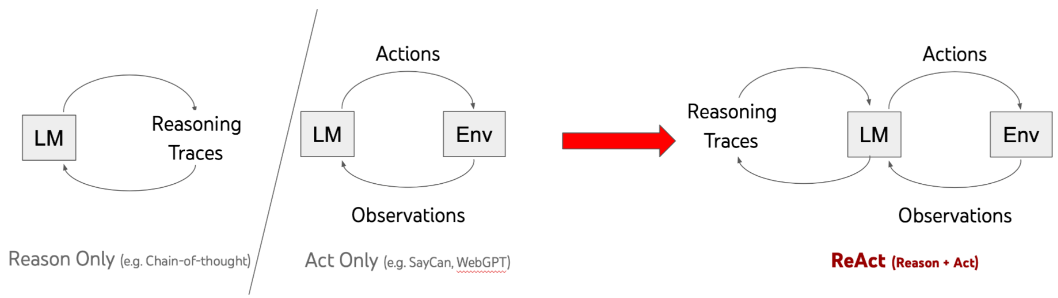
Actually… This is not entirely new

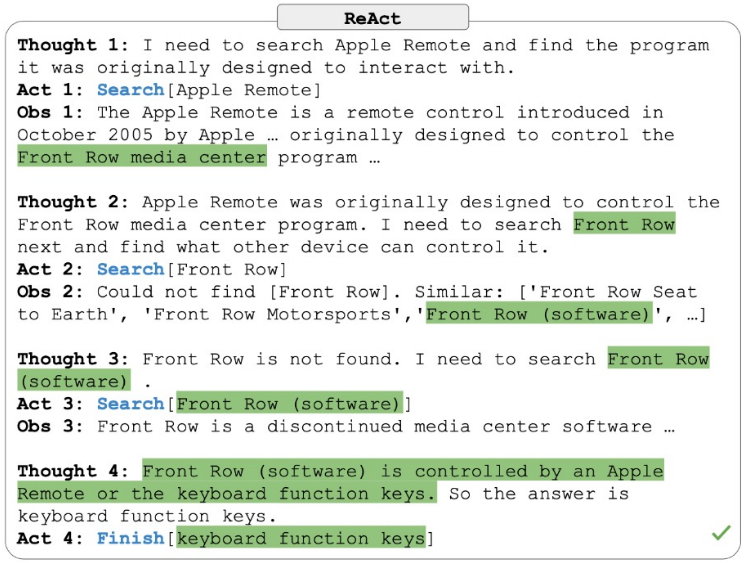
- **AutoGen** does code generation and execution as part of conversation
  - But it doesn’t natively use ReAct framework and can result in a very long-winded conversation

- **Transformers Agents v2** has ReAct Code Agent in addition to ReAct JSON Agent (https://huggingface.co/blog/agents)
  - But prompt used is really long and not targeted (see Additional Slides)

Recap: ReAct (Reasoning and Acting) Framework

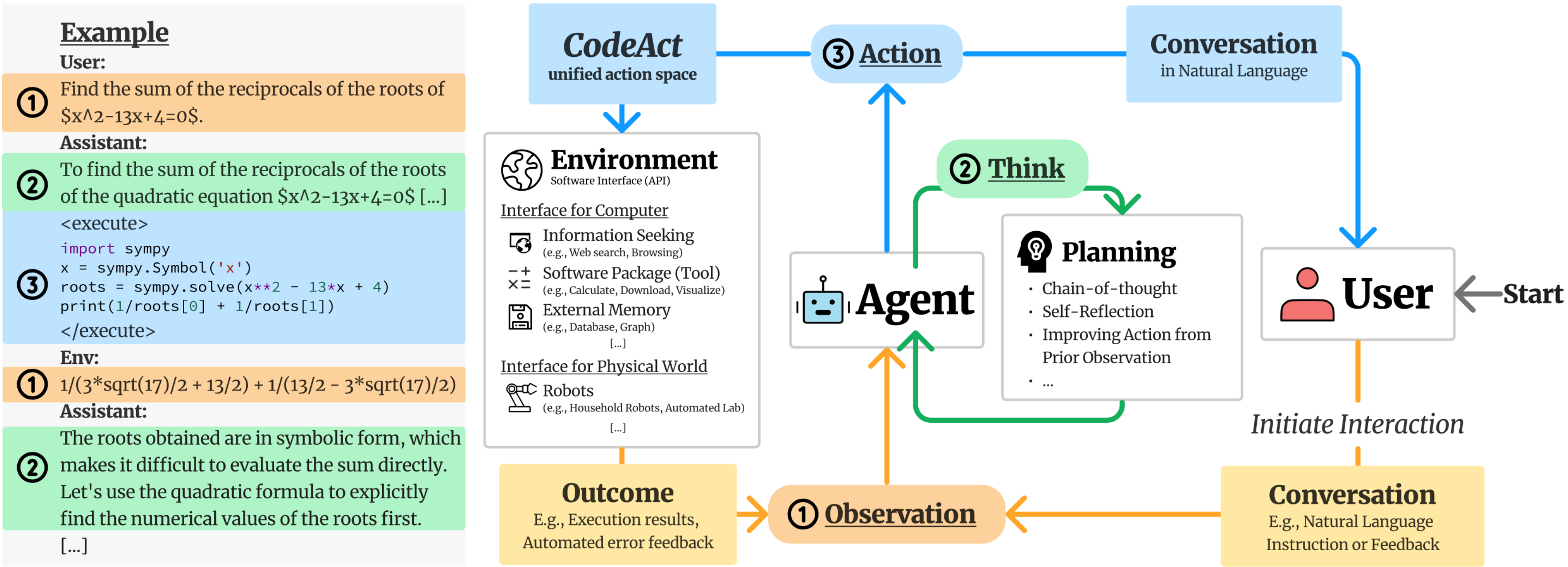
- LLMs do not have access to the environment - Using **Observations**, align LLMs to the environment
- LLMs may not be good at one-step reasoning – Using **Thoughts**, generate action plan





ReAct: Synergizing Reasoning and Acting in Language Models. Yao et al. 2022

ReAct with Code as Action



CodeAct Agent’s 4 steps

- **Thoughts**
  - What agent thinks it needs to do
- **Code / Final Answer**
  - What code to execute what it needs to do
  - This part can be outputting final answer too
- **Observation**
  - Output of the code / errors (if any)
  - This is given by the environment and not generated by the agent
- **Expert Feedback based on Trajectory** (Optional)
  - Expert takes in Task and Agent Trajectory and guides Agent on whether the action is correct

CodeAct Prompt

You are a helpful assistant assigned with the task of problem-solving. To achieve this, you will be using an interactive coding environment equipped with a variety of tool functions to assist you throughout the process.

At each turn, you should first provide your step-by-step thinking for solving the task. Your thought process should be enclosed using "**<thought>**" tag, for example: **<thought>** I need to print "Hello World!" **</thought>**.

After that, you have two options:

1) Interact with a Python programming environment and receive the corresponding output. Your code should be enclosed using "**<execute>**" tag, for example: **<execute>** print("Hello World!") **</execute>**.

2) Directly provide a solution that adheres to the required format for the given task. Your solution should be enclosed using "**<solution>**" tag, for example: The answer is **<solution>** A **</solution>**.

You have **{max_total_steps}** chances to interact with the environment or propose a solution. You can only propose a solution **{max_propose_solution}** times.

{tool_desc}

{in_context_example}

{task_prompt}

CodeAct Expert Feedback prompt

You are an expert tasked with evaluating and providing feedback on an assistant's performance.

Here is an example. Please follow the format as the following expert acts.

{in_context_example}

{tool_desc}

{trajectory}

{correct_solution}

Please provide concise and constructive feedback. Remember, your role is similar to a teacher. Rather than giving away the solution or details about the answer, guide the assistant toward understanding how to arrive at the correct answer. Your feedback should focus on enhancing the assistant's ability to think critically and respond accurately. Now provide your feedback.

Expert feedback:

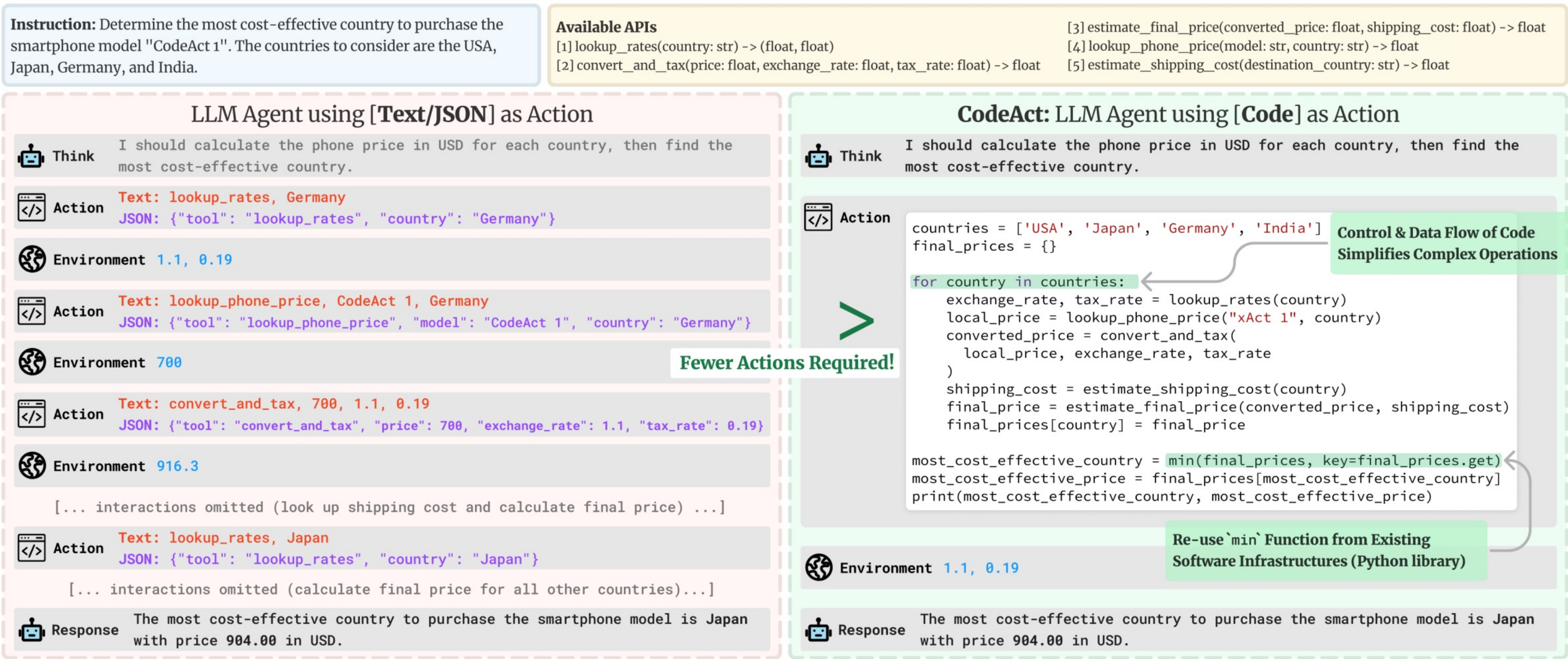
The three kinds of action formats

Table A.6: Example of actions for re-purposed API-Bank (Li et al., 2023) and M^3ToolEval.

Format	Action
`CodeAct`	`AddAgenda(content="Meeting with John", time="2023-10-26 09:00:00")`
JSON	`{"action": "AddAgenda", "content": "Meeting with John", "time": "2023-10-26 09:00:00"}`
Text	`Action: AddAgenda, content: Meeting with John, time: 2023-10-26 09:00:00`

Note: JSON and Text can be easier to check for input parameter match. CodeAct is tougher because it could be variable names instead of actual input value.

Text/JSON vs Code



Note there is no correctness check of parameter and types for function inputs in CodeAct

Text/JSON

Table 1: The benefit of `CodeAct` compared to using Text/JSON for LLM action.

	`CodeAct` for LLM action	**JSON or Text for LLM action**
Availability of Data	✔Large quantity of code available[1] for pre-training	✗Data curation required for particular format
Complex Operation (e.g., looping, composition of multiple tools)	✔Natively supported via control and data flow	✗Requires careful engineering if feasible (e.g., define new tools to mimic if-statement)
Availability of Tools	✔Can directly use existing software packages[2]	✗Requires human effort to curate tools from scratch or existing software
Automated Feedback	✔Feedback mechanism[3] (e.g., traceback) is already implemented as an infrastructure for most programming languages	✗Requires human effort to provide feedback or re-route feedback from the underlying programming language used to implement the tools

[1] Including code demonstrating useful behaviors for LLM agents (e.g., task decomposition, coordination of multiple function calls to different tools).
[2] Human-written Python packages covering a wide range of applications are available on `https://pypi.org/`.
[3] For example, in Python, errors and exceptions (`https://docs.python.org/3/tutorial/errors.html`) are available. Most software provides error messages in natural language to help human programmers debug their code. `CodeAct` enables LLM to use them directly.

Why is code better?

- Enables more complex process flows (for, while, if loops)
- Stores intermediate state within code without need for Agent to know

- My take: Typically better if Agent is not good at planning and handling multiple intermediate steps of plan
  - Caveat: Will be harder to error-correct multiple steps at a time in code

<u>Code</u>

```
for person in GetPersonList(org):
    print(GetAddress(person))
```

<u>Modular Function Calling</u>

GetPersonList(org)

-> Returns Person1, Person2, Person3 etc.

GetAddress(Person1)

GetAddress(Person2)

GetAddress(Person3)

Recap: ARC Challenge

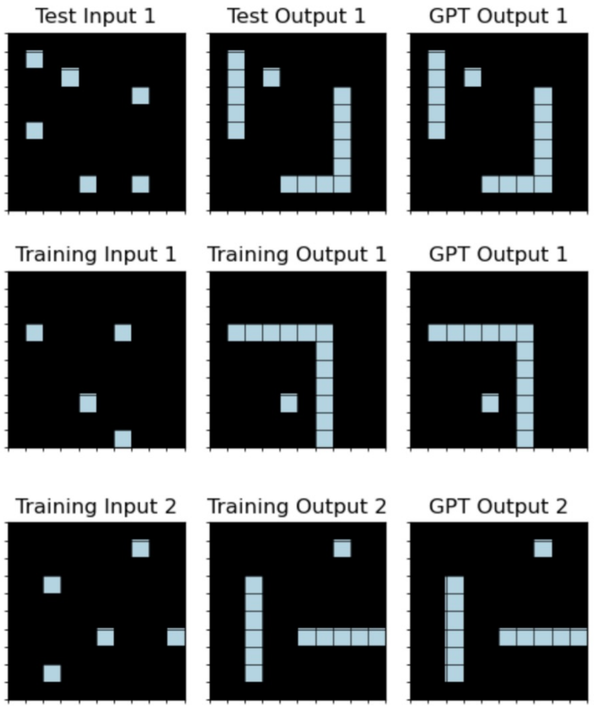
Conditional Code

```
def transform_grid(grid):
        objects = get_objects(grid)
        new_grid = [list(row) for row in grid]
        for obj1 in objects:
                for obj2 in objects:
                        if on_same_line(obj1['tl'], obj2['tl'], 'row'):
                                new_grid = fill_line_between_coords(new_grid, obj1['tl'],
obj2['tl'], obj1['value'])
                        if on_same_line(obj1['tl'], obj2['tl'], 'col'):
                                new_grid = fill_line_between_coords(new_grid, obj1['tl'],
obj2['tl'], obj1['value'])
        return [''.join(row) for row in new_grid]
```

Michael Hodel's DSL
https://github.com/michaelhodel/arc-dsl

```
def solve_ded97339(I):
    x1 = ofcolor(I, EIGHT)
    x2 = product(x1, x1)
    x3 = fork(connect, first, last)
    x4 = apply(x3, x2)
    x5 = fork(either, vline, hline)
    x6 = mfilter(x4, x5)
    O = underfill(I, EIGHT, x6)
    return O
```

ARC Training Set
ded97339.json



Even for single API calls, CodeAct yields improvements over Text/JSON for some models

Table 2: Atomic API call correctness on API-Bank. The best performance is **bolded**, and the second-best is <u>underlined</u>.

Format of Action	Correctness (%, ↑) CodeAct	JSON	Text
Open-source LLMs			
CodeLlama-7b-Instruct-hf	<u>12.5</u>	12.0	**17.0**
CodeLlama-13b-Instruct-hf	<u>11.8</u>	7.8	**14.0**
CodeLlama-34b-Instruct-hf	**17.3**	12.0	<u>16.8</u>
Llama-2-7b-chat-hf	**28.8**	11.3	<u>25.8</u>
Llama-2-13b-chat-hf	**38.1**	8.5	<u>37.3</u>
Llama-2-70b-chat-hf	<u>35.6</u>	14.3	**37.6**
Mistral-7B-Instruct-v0.1	<u>2.5</u>	2.3	**3.0**
lemur-70b-chat-v1	**58.6**	46.6	<u>56.1</u>
Closed-source LLMs			
claude-2	**76.7**	59.4	<u>73.7</u>
claude-instant-1	**75.2**	64.9	<u>73.2</u>
gemini-pro	70.4	**73.2**	<u>71.2</u>
gpt-3.5-turbo-0613	**74.4**	<u>73.9</u>	73.4
gpt-3.5-turbo-1106	<u>75.4</u>	**78.4**	73.4
gpt-4-0613	<u>75.4</u>	**82.0**	74.4
gpt-4-1106-preview	<u>76.7</u>	**82.7**	73.4
text-davinci-002	**69.2**	<u>59.6</u>	57.4
text-davinci-003	<u>75.4</u>	**76.9**	69.7

JSON for Function Calling is still better for gpt-4 series
StrictJSON / TaskGen will do well for those ☺

Can CodeAct self-learn?

- Not really, it is still limited to what the multiple rounds of thoughts and self-reflection/expert feedback can do

- My take: For true self-learning, we need to have **more than one agent** in the world, all learning from one another and **having different initial biases / memory**. Fine-tuning / updating of memory will then be the basis of self-learning

Questions to Ponder

- When would using code as actions be disadvantageous?
- How can type checking and input parameter checking be still done for code?
- Is there a way to emulate the good properties of using code for planning, but do it natively in the Agent's planning?
- How can intermediate states be better stored in native Function Calling-like interfaces?

Additional Slides

CodeAct Task-Code Example (Alfworld)

Task (Alfworld):

You are in the middle of a room. Looking quickly around you, you see a cabinet 13, a cabinet 12, a cabinet 11, a cabinet 10, a cabinet 9, a cabinet 8, a cabinet 7, a cabinet 6, a cabinet 5, a cabinet 4, a cabinet 3, a cabinet 2, a cabinet 1, a coffeemachine 1, a countertop 1, a diningtable 1, a drawer 1, a fridge 1, a garbagecan 1, a microwave 1, a shelf 3, a shelf 2, a shelf 1, a sinkbasin 1, a stoveburner 4, a stoveburner 3, a stoveburner 2, a stoveburner 1, and a toaster 1.

Your task is to: put a clean lettuce in diningtable.

Code (first part to find lettuce):

```
for loc_to_check in ["fridge 1", "diningtable 1", "sinkbasin 1", "stoveburner 1", "stoveburner 2", "stoveburner 3", *[f"cabinet {i}" for i in range(1, 14)]]:
    res = goto(loc_to_check)
    if "closed" in res:
        res = open_receptacle(loc_to_check)
    if "lettuce" in res:
        print(res)
        print(f"Found a lettuce in {loc_to_check}.")
        break
```

Transformers Agents v2 Prompt (Part 1)
-> A lot more verbose than CodeAct

- You will be given a task to solve as best you can.
- You have access to the following tools: **<<tool_descriptions>>**
- To solve the task, you must plan forward to proceed in a series of steps, in a cycle of 'Thought:', 'Code:', and 'Observation:' sequences
- At each step, in the 'Thought:' sequence, you should first explain your reasoning towards solving the task, then the tools that you want to use
- Then in the 'Code:' sequence, you should write the code in simple Python. The code sequence must end with '/End code' sequence
- During each intermediate step, you can use 'print()' to save whatever important information you will then need.
- These print outputs will then be available in the 'Observation:' field, for using this information as input for the next step.
- In the end you have to return a final answer using the `final_answer` tool.

https://huggingface.co/docs/transformers/v4.41.2/en/main_classes/agent#transformers.ReactAgent

Transformers Agents v2 Prompt (Part 2)
-> A lot more verbose than CodeAct

Task: "Generate an image of the oldest person in this document.”

Thought: I will proceed step by step and use the following tools: `document_qa` to find the oldest person in the document, then `image_generator` to generate an image according to the answer.

Code:```python

answer = document_qa(document=document, question="Who is the oldest person mentioned?")

print(answer)

```<end_code>

Observation: "The oldest person in the document is John Doe, a 55 year old lumberjack living in Newfoundland.”

Thought: I will now generate an image showcasing the oldest person.

Code:

```python

image = image_generator("A portrait of John Doe, a 55-year-old man living in Canada.")

final_answer(image)

```<end_code>

https://huggingface.co/docs/transformers/v4.41.2/en/main_classes/agent#transformers.ReactAgent
```