① Deep Agents from Scratch (by LangChain Academy)

- Earlier LLMs were good for generating text and could reason about information presented in their input context
  → single pass operations like RAG or chat
- Today LLMs can be trained to generate structured output, such as tool calls
  → interact with traditional computing infrastructure
  → build an agent (e.g. ReAct agent that reasons and acts)
    - Operates in a loop (tool call → feedback → repeat if necessary).
- Long-running agents: Claude Code, Deep Research, Manus
  - planning tool
  - file system
  - subagents
  - prompting (detailed)
- Earlier, lots of logic needed to be implemented
- Now, the logic has moved into the prompt and the machinery surrounding the LLM has become simpler
- Deep agent:
  - long-running } agents
  - highly capable }
  - https://github.com/langchain-ai/deepagents
    - Planning tool
    - File system
    - Sub agents
    - system prompt
    - pip install deepagents (tavily-python)
    - LangGraph graph (streaming, HITL, memory, studio)
    - tools (req.), instructions (req.), subagents (opt.), model (opt.), builtin_tools (opt.)
    - Supports: async, MCP, configurations
- Python 3.11+ w/ uv package manager

(2)

- uv venv --python 3.11
  uv sync
  uv run jupyter notebook (localhost:8888)
- Environment variables: LangSmith, OpenAI/Anthropic, Tavily

- LangChain Academy courses: https://academy.langchain.com/courses/deep-agents-with-langgraph
- Common deep agent approaches
  - Task planning (e.g. todo), often with recitation
  - Context offloading to file systems
  - Context isolation through sub-agent delegation

Module 1: Course Overview

- Agents are working on
  1) more general tasks
  2) over longer time horizons
- Deep agent may do tens of tool calls or turns
- 4 central principles common for deep agents:
  1) planning (todo list, updates), approval for plan, start work
  2) offload context (file system), read/write, long-term memory
  3) Task delegation (sub-agents), context isolation, careful w/ conflicts, parallelization + summary
  4) careful / extensive prompt engineering.

LangGraph state also ok

- Deep agents on GitHub: https://github.com/langchain-ai/deepagents

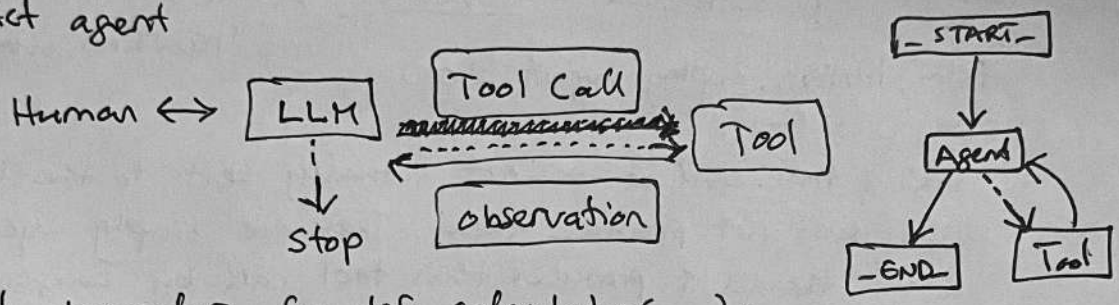
Module 2: Create Agent

- Create ReAct agent (pre-built)
- 0_create_agent.ipynb
- ReAct agent is an AI agent that uses the "Reasoning and Acting" (ReAct) framework to combine chain-of-thought (CoT) reasoning with external tool use.
- Three components (this agent): LLM, tools, prompt (context)
- create_react_agent → create_agent: key capabilities
  - Memory integration (short-term & long-term)
  - Human-in-the-loop control: interrupt execution (human review)
  - Streaming support: agent state, model tokens, tool outputs, combined stream
  - Deployment tooling: LangGraph Platform supports testing, debugging, and deployment
    - Studio (UI), LangSmith (observability),

③

- Simple agent: agent - tool loop
- Complex agent: — " — + customized with pre/post hooks and/or structured outputs

- ReAct agent



- @tool decorator for def calculator(...):
- from langgraph.prebuilt import create_react_agent
  from langchain.chat_models import init_chat_model
  model = init_chat_model(model="openai:gpt-4o-mini", temperature=0.0)
  tools = [calculator]
  agent = create_react_agent(model, tools, prompt=SYSTEM_PROMPT)
  .with_config({"recursion_limit": 20})

  (compiled-stategraph)
  → defaults to AgentState (messages)
  even more available

```
from IPython.display import Image, display
display(Image(agent.get_graph(xray=True).draw_mermaid_png()))
result1 = agent.invoke({"messages": [{"role": "user",
                                      "content": "what is 3.1*4.2?"
                                     }]
                       })
from utils import format_messages
format_messages(result1["messages"])
```

HumanMessage → AIMessage → ToolMessage → AIMessage
(tool call) (tool output)

- LangGraph state can be shared between Nodes in the graph

AgentState (default) - remaining_steps: NotRequired[RemainingSteps] ← recursion limit
- messages: Annotated[Sequences[BaseMessage], add_messages]

- Custom state: CalcState(AgentState):
  ops: Annotated[List[str], reduce_list]
  - extend calculator's AgentState with "ops"
  - injected state: not sent to LLM

```
def calculator_wstate(operation, a, b,
        state: Annotated[CalcState, InjectedState]
        tool_call_id: Annotated[str, InjectedToolCallId])
```

- update state: return Command(update={"ops": ops, "messages": [ToolMessage(...)]}) directly

④ - agent = create_react_agent(
model,
tools,
prompt = SYSTEM_PROMPT,
!!! state_schema = CalcState).with_config(
{"recursion_limit": 20})

- from IPython.display import JSON
JSON(result2)
- state & tool_call_id are not actually sent to the LLM. LLM does not produce them. They are simply injected after the LLM produces this tool call by LangGraph. This gives tool node access to them when it's actually invoking the tool but doesn't burden the LLM itself with having to produce those arguments.
- Command : goto & update simultaneously

• create_react_agent(before END structured output "structured_response" called before agent node →e.g. summarize messages
model, tools, *,
Optional: prompt, response_format, pre_model_hook,
post_model_hook, state_schema, context_schema,
call after LLM call but before tool call →e.g. HITL, post-processing ← checkpointer, store, interrupt_before,
interrupt_after, debug, version, name,
**deprecated_kwargs) → Compiled StateGraph:

Module 3 : Todo Lists

- planning component / tool
- steer long-running, complex tasks → structured todo lists before executing tasks
- Basic pre-built AgentState has messages
→ add todos (content + status) and files (virtual file system)
- A long-running agent can use a todo list to stay on task
→ tools: write_todo, read_todo
- List contains individual tasks.
→ update with a full rewrite of the list
→ allows LLM to reconsider tasks as it makes progress
- from deep_agents_from_scratch.prompts import WRITE_TODOS_DESCRIPTION
(system prompt on how to manage todos)
create and
- @tool(description = WRITE_TODOS_DESCRIPTION)
def write_todos(todos: list[Todo], todo_call_id: Annotated[str, InjectedToolCallId]) → Command

⑤

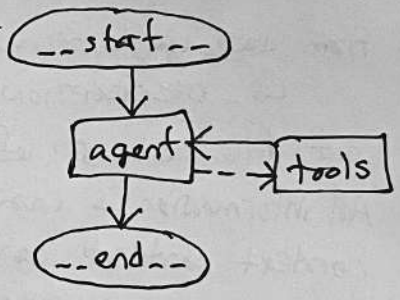
```
        return Command(update={
            "todos": todos,
            "messages": ToolMessage(f"Updated todo list to {todos}",
                                    tool_call_id=tool_call_id)
        })

@tool(parse_docstring=True)
def read_todos(state: Annotated[DeepAgentState, InjectedState],
               tool_call_id: Annotated[str, InjectedToolCallId]) -> str:
    ...
    return result(string)
```

→ ~~Create~~ Create ReAct agent will package that string into a tool message and update the messages field in state with the tool observation

- follow Marcus approach of having a todo recitation after each task

```
from deep_agents_from_scratch.prompts import TODO_USAGE_INSTRUCTIONS
(system prompt)
...
tools = [write_todos, web_search, read_todos]
prompt = TODO_USAGE_INSTRUCTIONS ... SIMPLE_RESEARCH_INSTRUCTIONS
```



- write_todos tool call contain todos list which will be stored to the agent's state
- web_search tool call contains result to a web query
- read_todos tool call lists all todos in the state
- write_todos tool call will finally update the todos list by marking pending items as completed
- finally AI is ready to provide the final result/output to the user
- LangGraph state is a typed data structure
  - → available to each node of the graph for its duration (also possible to store for long-term storage)
  - → data types & reducer function (how info is add to the element)
    - → especially useful when a task is mapped to multiple nodes, which are executed in parallel and update state simultaneously
- typing.Annotated allows to attach arbitrary metadata to a type hint
  Annotated[type, metadata1, metadata2, ...]
- remaining_steps tracks steps in a graph (recursion_limit)
- create_react_agent
  - pre-hook, post-hook → e.g. summarize (node before) / e.g. HITL, guardrails, validation, postprocessing (node after)
  - response_format (node before END) structured output (structured response)

⑥ • InjectedState provides the tool access to the graph state
• Command updates values in state + redirect to next node
• Prompt for using todos (for the agent)

Module 4: Files

- Agent context window can grow rapidly during complex tasks
  → offload context through file system operations
  → save information to files and fetch it as-needed
- Mock file system: key = file path, value = file content
  → short-term, thread-wise persistence
  → single-agent conversation
  → NOT suited for information that needs to persist across different conversation threads

- tools: ls, read_file, write_file
- Write data to a file, read some or all from the file later
- from deep_agents_from_scratch.prompts import (
  LS_DESCRIPTION, READ_FILE_DESCRIPTION, WRITE_FILE_DESCRIPTION)
  def file_reducer(left, right):
- All information is easily kept in context, but for long-running agents context content can be compressed or eliminated. Storing information prior to compression and retrieving it when it's needed is smart context engineering.
- system prompt for the agent: you have three tools: xxx to manage a virtual file system
- Built on top of a mock file system in LangGraph state

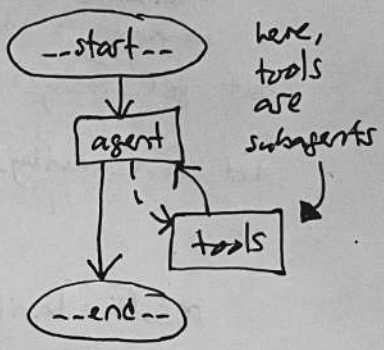
Module 5: Subagents

- Specialized agents → context isolation
- Issues with long context
  - context clash
  - context confusion
- Context isolation provides an effective solution by delegating tasks to specialized sub-agents, each operating within their own isolated context window.
  → prevents context clashes, confusion, poisoning, and dilution

(7)

→ enables focused, specialized task execution

- Subagents with different tool sets tailored for specific tasks
  - → registry dictionary with subagent_type as the key
  - → subagent operates in complete isolation from the parent's context → clean separation of concerns
- SubAgent(TypedDict):
  - name: str
  - description: str
  - prompt: str
  - tools: NotRequired[list[str]]
- Task delegation tool for context isolation through sub-agents.
  - → generate sub-agents as tools
  - → subagents act as both tools and agents
- Supervisor agent
  - subagent 1
  - subagent 2
  - ...



- supervisor agent's prompt
  - task: coordinate to subagents
  - available_tools: task, think
  - hard_limits: iterations, focus
  - scaling_rules: instructions for use ~~with~~ examples
- Do not give a single agent a lot of tools → gets confused!

Module 6: Full Agent

- LangGraph > create_react_agent → LangChain (v1.0) > create_agent
- save raw tool observation to reproduce, use summary in tool calls (token heavy)
- Tavily: search engine that is well designed for AI applications
  1) search execution using Tavily API
  2) content summarization using GPT-4o-mini
  3) Result processing using markdownify to convert ~~the~~ HTML to markdown
  4) context offloading saves to file
  5) strategic thinking provides structured reflection
- search tool that offloads raw contents to files and returns only a summary to the agent

(8) → solves token efficiency problem by storing detailed search results in files while keeping the agent's working context minimal and focused.

```
• import httpx
from markdownify import markdownify
from pydantic import BaseModel, Field
from tavily import TavilyClient
from deep_agents_from_scratch.prompts import SUMMARIZE_WEB_SEARCH
from deep_agents_from_scratch.state import DeepAgentState

summarization_model = init_chat_model(model="openai:gpt-4o-mini")
tavily_client = TavilyClient()
class Summary(BaseModel):
    filename: str = ...
    summary: str = ...
def get_today_str() -> str:
    ...
def run_tavily_search(search_query: str, max_results: int = 1,
                      topic: Literal["general", "news", "finance"] = "general",
                      include_raw_content: bool = True) -> dict:
    result = tavily_client.search(search_query, max_results=max_results,
                                  include_raw_content=include_raw_content,
                                  topic=topic)
    return result
def summarize_webpage_content(webpage_content: str) -> Summary:
    structured_model = summarization_model.with_structured_output(Summary)
    summary_and_filename = structured_model.invoke(
        [HumanMessage(content=SUMMARIZE_WEB_SEARCH.
            format(webpage_content=webpage_content,
            date=get_today_str()))
        ])
    return summary_and_filename
def process_search_results(results: dict) -> list(dict):
    processed_results = []
    HTTPX_CLIENT = httpx.Client()          → Tavily results
    for result in results.get('results', []):
        ...
        processed_results.append({'url': result['url'],
                                  'title': result['title'],
                                  'summary': summary_obj.summary,
                                  'filename': summary_obj.filename,
                                  'raw_content': raw_content})
    return processed_results
```

⑨

```
@tool(parse_docstring=True)
def tavily_search(query: str, state: ...) -> Command:
    # search web and save detailed results to files while
    # returning minimal context. (context offloading)
    search_results = run_tavily_search(...)
    processed_results = process_search_results(search_results)
    # save each result to a file and prepare summary
    # create minimal summary for tool message
    return Command(update={"files": files, "messages": [ToolMessage(
        summary_text, tool_call_id=tool_call_id)]})
```

FINAL ▽ TOOL

```
@tool(parse_docstring=True)
def think_tool(reflection: str) -> str:
    return f"Reflection recorded: {reflection}"
```

STOP & REFLECT FOR LLM

Used after each search to analyze results and plan next steps systematically.

- Tavily search → HTTPX.get(url) → markdownify(response.text)
  → summarize_webpage_content(raw_content)
- Only summaries are provided to messages, offloading raw content to files
  → use read_file() to access full details when needed
- Deep Agent

```
model = init_chat_model(model="anthropic:claude-sonnet-4-20250514",
                        temperature=0.0)

max_concurrent_research_units = 3
max_researcher_iterations = 3
sub_agent_tools = [tavily_search, think_tool]
built_in_tools = [ls, read_file, write_file, write_todos, read_todos,
                  think_tool]

research_sub_agent = {"name": "research-agent",
                      "description": "Delegate research to a subagent
                          researcher. Only give this researcher
                          one topic at a time.",
                      "prompt": RESEARCHER_INSTRUCTIONS.format(
                          date=get_today_str()),
                      "tools": ["tavily_search", "think_tool"]}
task_tool = _create_task_tool(sub_agent_tools, [research_sub_agent],
                              model, DeepAgentState)
delegation_tools = [task_tool]
all_tools = sub_agent_tools + built_in_tools + delegation_tools
```

(10)

```
SUBAGENT_INSTRUCTIONS = SUBAGENT_USAGE_INSTRUCTIONS.format(
    max_concurrent_research_units = max_concurrent_research_units,
    max_researcher_iterations = max_researcher_iterations,
    date = datetime.now().strftime("%a %b %-d, %Y"))
```

- Research Agent Prompt (instructions) } RESEARCHER_INSTRUCTIONS
  1) Role
  2) task
  3) available tools
  4) instructions
  5) hard limits
  6) show your thinking prompt
- Supervisor (Deep Agent) (instructions) } INSTRUCTIONS
  1) Todo management
  2) File system usage
  3) subagent delegation
     - 3.1) Task
     - 3.2) Available tools
     - 3.3) Hard limits
     - 3.4) Scaling rules

```
agent = create_react_agent(model, all_tools, prompt = INSTRUCTIONS,
                           state_schema = DeepAgentState)
result = agent.invoke({"messages": [{"role": "user",
                                     "content": "Give me an overview of MCP"}]})
```

same

- deepagents package: https://github.com/langchain-ai/deepagents
  - file system tools (context offloading)
  - todo tool (planning)
  - task tool

→ only supply subagent and any tools you want to use

```
agent = create_deep_agent(sub_agent_tools, INSTRUCTIONS,
                          subagents = [research_sub_agent],
                          model = model)
result = agent.invoke(...)
```

- Deep agents
  - planning: prior to task execution
  - computer access: shell & file system
  - subagent delegation: isolated task execution

Module 7: Conclusion

(11) · Deep Agents
1) Planning Tool
2) File System
3) Subagents
4) Prompts (System)

· Deep Agents UI
- UI for interacting with deep agents