

0

3

SECTION 3

Deep Agents 1.0

Batteries-included agent harness: planning tool, virtual filesystem,
subagents with isolated context, pluggable backends,
HITL middleware.

Built on LangGraph + LangChain 1.0 middleware.
Inspired by Claude Code,
Deep Research, Manus.

LangGraph limits: why a new layer

WARNING

LangGraph is a runtime, not an agent harness.

You still have to wire planning, filesystem access, subagent isolation, HITL approval, and context offload yourself.

PROS

- + Full control over graph topology, cycles, parallel branches (Send)
- + Durable execution, checkpointing, interrupt-based HITL are first-class
- + Stable public API until 2.0 (released 22 Oct 2025)

CONS

- Boilerplate-heavy: planning tool, fs tools, subagent wiring -- all manual
- No built-in context overflow strategy (every tool result floods the main thread)
- No opinionated "coding/research" agent defaults -- you re-implement Claude Code patterns

Deep Agents: opinionated defaults

INFO

Analogy: Django over raw WSGI

Same runtime (LangGraph), but with conventions and pre-built middleware.

```
// examples/hello.py:1-14
```

```
# One import, one call -- you get:
# - write_todos planning tool
# - ls / read_file / write_file / edit_file
# - glob, grep, execute (bash)
# - task tool for subagents
# - summarization middleware
```

```
from deepagents import create_deep_agent
```

```
>
> agent = create_deep_agent(
// note: snippet has 14 lines, card fits ~10
> model="openai:gpt-4.1",
> tools=[my_tool],
> system_prompt="...",
```

source: docs.langchain.com/oss/python/deepagents/overview

SUCCESS

Stack

LangGraph (runtime)
 -> create_agent (LangChain 1.0, thin harness)
 -> create_deep_agent (opinionated harness)

Built-in: planning, FS, subagents, HITL, skills.

pip install deepagents

// setup.sh:1-5

```
pip install deepagents
```

```
# or, with uv:
```

```
uv add deepagents
```

// note: snippet has 5 lines, card fits ~3

```
# JS analogue:
```

```
npm install deepagents
```

INFO

Latest stable (snapshot 2026-06-22)

Python: deepagents 0.6.11 (no formal 1.0 yet).

Repo: github.com/langchain-ai/deepagents (~24.9k stars)

WARNING

Dependencies

deepagents pulls in langchain, langgraph, langchain-core.

API keys via env vars: OPENAI_API_KEY, ANTHROPIC_API_KEY, etc.

create_deep_agent: hello world

// examples/hello.py:1-12

```
from deepagents import create_deep_agent

> agent = create_deep_agent(
>     model="openai:gpt-4.1",
>     tools=[],
>     system_prompt="You are a helpful assistant.",
> )

result = agent.invoke({
    "messages": "Write a haiku about Python"
})
print(result["messages"][-1].content)
```

INFO

What you get for free

- write_todos tool
- ls / read_file / write_file / edit_file
- glob, grep, execute (bash)
- task tool for subagents
- summarization middleware
- same LangGraph runtime as create_agent

create_deep_agent: full API

// examples/full_api.py:1-15

```
from deepagents import create_deep_agent
from deepagents.backends import FilesystemBackend

> agent = create_deep_agent(
>   model="anthropic:claude-sonnet-4-5",
>   tools=[my_tool],
>   system_prompt="...",
>   subagents=[researcher, writer],
>   skills=["./skills/review.md"],
>   backend=FilesystemBackend("./ws"),
>   middleware=[my_hitl, my_logger],
>   checkpointer=InMemorySaver(),
>   store=InMemoryStore(),
>   interrupt_on={"bash": True},
> )
```

System prompt: how to talk to a deep agent

```
// examples/system_prompt.py:1-16
```

```
SYSTEM_PROMPT = """
```

```
You are a senior backend engineer.
```

```
> WORKFLOW:
```

- > 1. Plan with write_todos before non-trivial work.
- > 2. Explore the codebase via ls / glob / grep first.
- > 3. Use edit_file for surgical changes, write_file for new files.
- > 4. Delegate research tasks to the "researcher" subagent.
- 5. Run tests via execute; never claim success without output.

```
CONSTRAINTS:
```

- > - Do not modify files outside ./src.
- > - Stop and ask the user if requirements are ambiguous.

```
"""
```

```
agent = create_deep_agent(model=..., system_prompt=SYSTEM_PROMPT)
```

Built-in fs + planning tools

INFO

Eight opinionated defaults, all active unless you override.

Planning, filesystem, search, shell, and subagent delegation -- one import, zero setup.

write_todos

plan / decompose a task into ordered steps

ls

list directory entries in the virtual FS

read_file

read one or many files with line offsets

write_file

create or overwrite a file in the FS

edit_file

surgical string-replace edits (matches all occurrences)

glob

find files by pattern (e.g. `**/*.py`)

grep

regex search across files with context

execute

run a shell command (sandboxed if a backend enforces it)

write_file and edit_file in action

```
// examples/fs_tools.py:1-11
```

```
# The agent calls these tools by name;  
# you describe the goal in the prompt.
```

```
> PROMPT = """  
> 1. Write src/hello.py with a greet(name) function.  
> 2. Use edit_file to add a docstring to greet().  
> 3. Use write_file to add tests/test_hello.py.  
"""
```

```
agent = create_deep_agent(model="openai:gpt-4.1")  
agent.invoke({"messages": PROMPT})
```

SUCCESS

Context overflow protection

Tool outputs larger than a threshold are offloaded to the virtual FS

and replaced with a path + summary in the message history. The agent can re-read specific portions on demand. This is what lets deep agents handle large repos without blowing the context window.

write_todos: explicit planning inside the graph

INFO

write_todos is just a tool, like any other.

The LLM emits a structured plan, the graph stores it in state["todos"],

```
// examples/write_todos.py:1-12
```

```
write_todos(  
> todos=[  
>     {"content": "Read repo structure", "status": "in_progress",  
>      "activeForm": "Reading repo structure"},  
>     {"content": "Implement greet()", "status": "pending",  
>      "activeForm": "Implementing greet()"},  
>     {"content": "Add pytest cases", "status": "pending",  
>      "activeForm": "Adding pytest cases"},  
> ]  
)
```

// note: snippet has 12 lines, card fits ~10

```
# Status: pending | in_progress | completed  
# activeForm: present-continuous shown in UI
```

Plan, Act, Reflect loop

1. PLAN

write_todos:
- decompose task
- order steps
- mark in_progress



2. ACT

execute tool call
(read_file, edit_file, ...)
or delegate via task



3. REFLECT

update todo status
re-plan if blocked
log progress

loop until all todos = completed

INFO

The graph state carries the plan -- every LLM call sees it in the system message, so the agent self-monitors progress across turns.

task tool: delegate, isolate, return

INFO

Why subagents?

Long tool outputs and exploratory searches pollute the main thread.

```
// examples/task_call.py:1-10
```

```
# When the main agent emits a tool call like:
> task(
>   subagent_type="researcher",
>   description="Find papers on RAG evaluation",
>   prompt="Search arXiv for 2025-2026 RAG evaluation surveys.
>   Return a 150-word summary with 3 citations.",
> )

# Deep Agents spins up a fresh deep agent with the researcher profile,
# runs it to completion, and returns only the final message.
```

Subagents: minimal example

// examples/subagents.py:1-23

```
from deepagents import create_deep_agent
```

```
researcher = {
```

```
> "name": "researcher",
```

```
> "description": "Does deep web research, returns citations",
```

```
> "system_prompt": "You are a research specialist. Always cite sources.",
```

```
> "tools": [web_search], # optional, can be []
```

```
}
```

```
writer = {
```

```
> "name": "writer",
```

```
> "description": "Polishes prose into a final report",
```

```
> "system_prompt": "You are a writing specialist.",
```

```
> "tools": [],
```

```
}
```

// note: snippet has 23 lines, card fits ~16

```
agent = create_deep_agent(
```

```
    model="openai:gpt-4.1",
```

```
> tools=[],
```

```
source> subagents=[researcher, writer],s README
```

```
< )
```

Isolated context for subagents

MAIN AGENT

```
context = [  
  user task,  
  summary from researcher,  
  summary from writer  
]
```

task("researcher")

SUBAGENT: researcher

```
context = [  
  full web_search results,  
  all 14 sources,  
  notes, drafts, citations  
]  
return: 150-word summary
```

SUBAGENT: writer

```
context = [  
  researcher summary,  
  original user task  
]  
return: polished 200-word report
```

Virtual FS: state files dict

INFO

Files live in graph state, not on disk by default.

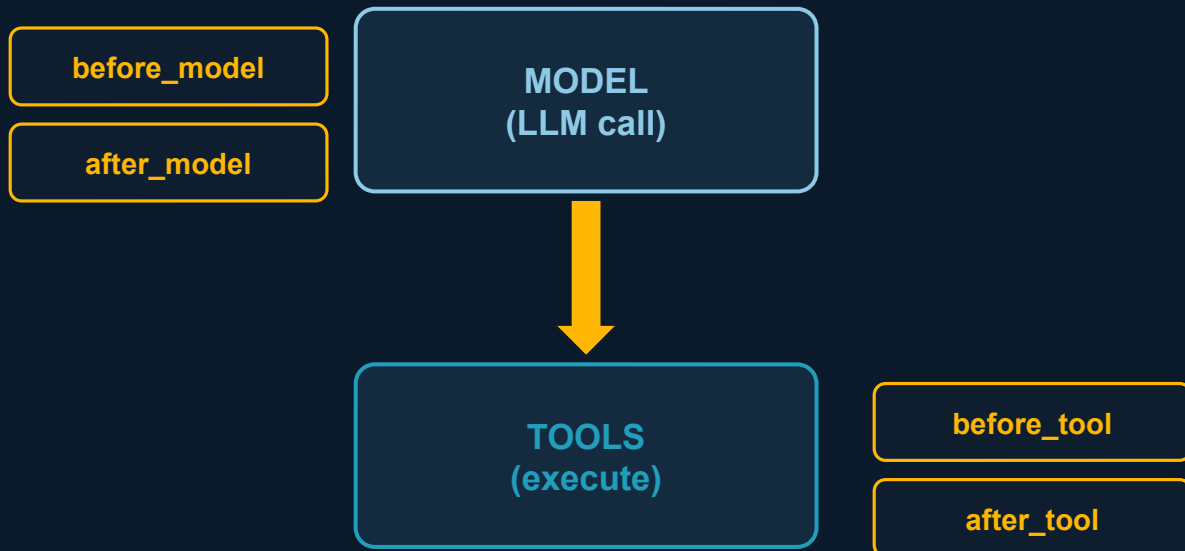
StateBackend keeps everything in state["files"]. Survives across turns in the same thread; never touches disk unless you swap backends

```
// examples/virtual_fs.py:1-9
```

```
# Inspect the virtual filesystem after a run:
result = agent.invoke({"messages": "Summarize repo"})
>
> files = result.get("files", {})
> for path, doc in files.items():
>     print(f"{path}: {len(doc.get('content', []))} bytes")

# /repo/src/main.py: 421 bytes
# /repo/README.md: 1804 bytes
```

Middleware: four hook points



INFO

Same middleware system as LangChain 1.0 `create_agent`.

Mix custom middleware with built-ins: Summarization, HumanInTheLoop, Filesystem, SubAgent.

Custom middleware: logging + PII redaction

// examples/middleware.py:1-18

```
from langchain.agents.middleware import AgentMiddleware
from deepagents import create_deep_agent

class LoggerMiddleware(AgentMiddleware):
> def before_model(self, state, runtime):
>     print(f"[model] {len(state['messages'])} msgs in")
>     return state
>
> def after_tool(self, state, runtime, tool_result):
>     print(f"[tool] {tool_result.tool_call_id} -> {len(str(tool_result.content))} chars")
    return state

agent = create_deep_agent(
    model="openai:gpt-4.1",
    middleware=[LoggerMiddleware(), HumanInTheLoopMiddleware(
>     interrupt_on={"execute": True}, # ask before running bash
< note: snippet has 18 lines, card fits ~16
    )],
> )
```

Pluggable backends: 4 flavors

StateBackend

default

Everything in graph state["files"]. Zero disk I/O.
Perfect for short-lived, sandboxed runs.

FilesystemBackend

local disk

Real directory on the host. Persists across runs.
Use when the agent should see/edit your repo directly.

StoreBackend

cross-thread

Lives in a LangGraph Store (Postgres, Redis).
Shared between threads and across sessions.

CompositeBackend

route by path

Route reads/writes to different backends depending on path prefix.
"/workspace" -> Filesystem, "/memory" -> Store.

CompositeBackend: route by path prefix

// examples/composite_backend.py:1-20

```
from deepagents import create_deep_agent
from deepagents.backends import (
    CompositeBackend, FileSystemBackend, StoreBackend
)
from langgraph.store.memory import InMemoryStore

store = InMemoryStore() # or PostgresStore in prod

backend = CompositeBackend(
    default=FileSystemBackend(root_dir="./workspace"),
    routes={
>     "/memory/": StoreBackend(store=store, namespace=("agent", "kb")),
>     "/scratch/": FileSystemBackend(root_dir="/tmp/scratch"),
> },
> )
```

// note: snippet has 20 lines, card fits ~16

```
agent = create_deep_agent(model=..., backend=backend)
# /workspace/notes.md -> local disk
# /memory/lessons.md -> Postgres, shared across sessions
```

source> # /scratch/tmp.py -> ephemeral tmpfs/backends

interrupt_on: approve tool calls

// examples/hitl.py:1-23

```
from langchain.agents.middleware import HumanInTheLoopMiddleware
from deepagents import create_deep_agent
from langgraph.checkpoint.memory import InMemorySaver
from langgraph.types import Command

agent = create_deep_agent(
    model="openai:gpt-4.1",
    checkpointer=InMemorySaver(),
    > middleware=[HumanInTheLoopMiddleware(
    > interrupt_on={
    > "execute": True, # bash -- always ask
    > "write_file": True, # disk writes -- always ask
    > "task": False, # subagents -- run unattended
    > },
    > ),
    > ],
```

//note: snippet has 23 lines, card fits ~15

```
cfg = {"configurable": {"thread_id": "user-42"}}
try:
```

source: `agent.invoke({"messages": ["deploy to staging"], cfg=cfg})`
 except InterruptedError:

stream_mode: tokens, updates, events

// examples/streaming.py:1-18

```
cfg = {"configurable": {"thread_id": "user-42"}}

# 1. Stream model tokens as they arrive
> for token, meta in agent.stream(
>   {"messages": "..."}, cfg,
>   stream_mode="messages",
> ):
    print(token.content, end="", flush=True)

# 2. Stream state updates per node
for chunk in agent.stream(
>   {"messages": "..."}, cfg,
>   stream_mode="updates",
> ):
    print(chunk) # {"model": {...}, "tools": {...}}

// note: snippet has 18 lines, card fits ~16
# 3. Subagent streams are surfaced as
#   {"subagent": {"name": "researcher", "chunk": ...}}
```

INFO

stream_mode

values

- "values": full state after each node
- "updates": delta per node (LangGraph-style)
- "messages": token-by-token LLM output
- "events": low-level LangGraph events
- "custom": writer().emit(...) from inside nodes

Tracing and evaluation: works out of the box

INFO

No code changes required.

Set `LANGSMITH_TRACING=true` plus `LANGSMITH_API_KEY` and `LANGSMITH_PROJECT`.

Every deep agent run is traced as a parent run with subagent runs nested underneath.

// examples/langsmith.sh:1-8

```
export LANGSMITH_TRACING=true
export LANGSMITH_API_KEY=lsv2_...
export LANGSMITH_PROJECT=deepagents-evals
```

```
python my_deep_agent.py
# -> all runs visible in smith.langchain.com
# -> subagent runs nested under the parent
# -> token usage, latency, tool errors captured
```

Real example: arXiv research agent

// examples/research_agent.py:1-24

```
from langchain.tools import tool
from deepagents import create_deep_agent
```

```
@tool
```

```
def arxiv_search(query: str, max_results: int = 5) -> str:
```

```
    """Search arXiv for papers matching the query."""
```

```
    import arxiv
```

```
    client = arxiv.Client()
```

```
    results = list(client.results(arxiv.Search(query=query, max_results=max_results)))
```

```
    return "\n\n".join(
```

```
        f"{r.title}\n{r.summary[:300]}..." for r in results
```

```
)
```

```
agent = create_deep_agent(
```

```
    model="openai:gpt-4.1",
```

```
    tools=[arxiv_search],
```

// note: snippet has 24 lines, card fits ~16

```
    system_prompt=( "You are a research assistant. Always cite paper titles and arXiv IDs."),
```

```
> subagents=[{
```

```
>     "name": "summarizer",
```

```
sou>     "description": "Compresses paper abstracts into a paragraph",
```

```
>     "system_prompt": "You are a precise summarizer."
```

Real example: coding agent, sandboxed

```
// examples/coding_agent.py:1-17
```

```
from deepagents import create_deep_agent
from deepagents.backends import SandboxBackend

> agent = create_deep_agent(
>   model="anthropic:claude-sonnet-4-5",
>   backend=SandboxBackend(
>     provider="daytona",    # or modal / runloop
>     api_key=os.environ["DAYTONA_API_KEY"],
>     image="python:3.12-slim",
>   ),
>   system_prompt=("You are a coding agent. Always run tests after edits. Stop and ask if requirements are ambiguous."),
> )

agent.invoke({"messages": "Add a /healthz endpoint to the FastAPI app, with tests."})

// note: snippet has 17 lines, card fits ~16
# Daytona/Modal/Runloop execute code in an isolated container;
# the local process never sees a stray rm -rf.
```


1.0-rc and the TypeScript port

INFO

What is new in 1.0-rc

- Full LangChain 1.0 middleware integration
- Stable Send API for parallel subagents
- Pluggable backends marked stable
- Skills: versioning + hot-reload
- CompositeBackend GA
- Note: as of snapshot 2026-06-22, latest stable is 0.6.11 -- 1.0 ships before EOY 2026

// examples/deepagentsjs.ts:1-13

```
// TypeScript analogue (deepagentsjs)
import { createDeepAgent } from "deepagents";
import { tool, z } from "@langchain/core/tools";

const search = tool(
  async ({ q }) => fetch("/api/search?q=" + q).then(r
=> r.text()),
  { name: "search", schema: z.object({ q:
z.string() }) },
);

> const agent = await createDeepAgent({
>   model: "openai:gpt-4.1",
>   tools: [search],
> });
```

When to choose Deep Agents

PROS

- + Batteries included: planning + FS + subagents + HITL + skills in one import
- + Less boilerplate than raw LangGraph, opinionated defaults that match Claude Code
- + Pluggable backends (local / Daytona / Modal / Runloop / LangSmith Store)
- + Skills system: reusable behaviors loaded on-demand
- + Open source (MIT), traceable through

CONS

- 1.0 not yet shipped (0.6.11 latest on snapshot 2026-06-22) -- breaking changes possible
- Opinionated: overriding defaults can be awkward
- Sandbox providers require external SaaS accounts
- Skills ecosystem is nascent, fewer ready-made skills than for Claude Code

INFO

Bridge to Open SWE

Open SWE (next section) is a real coding agent that uses Deep Agents as its harness. All the patterns from this section -- write_todos, subagents,

virtual FS, HITL -- show up unchanged in production at [langchain-ai/open-swe](https://langchain-ai.com/open-swe).

source: docs.langchain.com/oss/python/langgraph/agents/overview