// WHITE PAPER · PUBLISHED 5 NOVEMBER 2026

LangChain and LangGraph: What They Are, How They Work, and Why MAX Took Another Road

Two good libraries for building AI agents, why MAX passed on both, and the three ideas he kept

Jason Newell · MAX Research Collective · Report WPR-AGT-009 · Published 5 November 2026
Licensed under the Creative Commons Attribution 4.0 International License (CC BY 4.0), https://creativecommons.org/licenses/by/4.0/
Original: https://www.maxresearchcollective.com/articles/langchain-and-langgraph.html
Cite as: Newell, J. (2026). LangChain and LangGraph: What They Are, How They Work, and Why MAX Took Another Road. MAX Research Collective, WPR-AGT-009. https://www.maxresearchcollective.com/articles/langchain-and-langgraph.html

Two libraries, and where MAX landed

LangChain is the quickest way to get an AI agent working. LangGraph gives you control over how that agent runs in production. Both reached version 1.0 on October 22, 2025, and LangChain's agents now run on top of LangGraph. They're good tools, and we studied them closely while designing MAX, the Collective's AI Resident.

MAX isn't built on either one. He talks with people, and from the end of their sentence to the start of his answer he has under two seconds, which leaves no room for an extra layer. He's also a single Rust program with one writer on his memory. His medical sibling, MARA, needs every step in an audit chain the Collective owns, and a framework's internal steps would sit outside it. Three of LangGraph's ideas came home anyway, rewritten as plain code: draw the workflow before it runs, save state after every step, and pause for a person where money or the real world is at stake.

What they are

Both are free, open-source libraries from the company LangChain, available in Python and in JavaScript and TypeScript. They tackle two halves of one problem: getting a large language model (LLM), the kind of model behind ChatGPT and Claude, to do real work with tools and data.

- LangChain is the high-level agent framework. One call, create_agent, gives you a working agent that can talk to any major model provider (OpenAI, Anthropic, Google and others) and call your tools. It also ships more than 100 integrations for models, vector databases and document loaders.
- LangGraph is the low-level runtime underneath. You describe your application as a graph: boxes (nodes) that do work, arrows (edges) that decide what runs next, and a shared state object that every step reads and writes. LangGraph saves that state after each step, so a run can pause for a person, or survive a crash, and carry on exactly where it stopped.
- LangSmith is the paid companion platform for tracing, evaluation and deployment. It's optional. Most production teams use it anyway.
- Deep Agents is a newer, opinionated "harness" built on both, for long-horizon agents that plan, use a file system and start sub-agents of their own.

LangChain's own guidance is short. Use LangChain to ship standard agent patterns fast, and drop down to LangGraph when you need custom control flow, long-running processes or human oversight. LangChain agents already run on LangGraph. Moving from one to the other takes no rewrite.

How the stack fits together

Deep Agents uses LangChain's create_agent, which compiles into a LangGraph graph. You can work at whichever layer fits the job, and LangSmith traces all three.

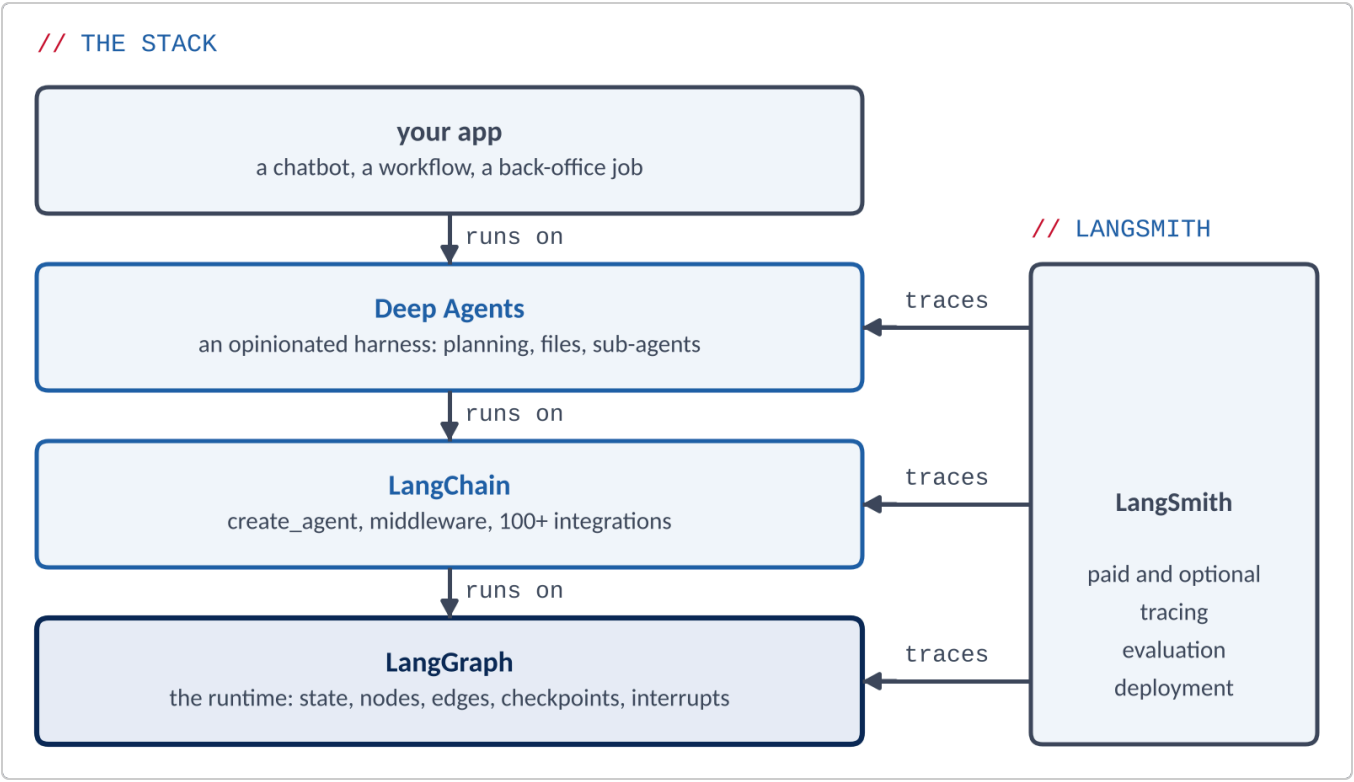


Figure 1. Each layer runs on the one below it, and LangSmith traces the three framework layers from the side.

How LangChain works

You build an agent by handing create_agent a model, a list of tools and a system prompt, the standing instruction the model reads before every conversation. Then the agent loops. The model reads the conversation and decides whether to call a tool; it sees the tool's result, and it goes round again until it can answer.

```
from langchain.agents import create_agent

def get_weather(city: str) -> str:
    """Get the current weather for a city."""
    return f"It's 54°F and raining in {city}."

agent = create_agent(
    model="anthropic:claude-sonnet-4-5",
    tools=[get_weather],
    system_prompt="You are a helpful assistant.",
)
```

```
result = agent.invoke(
    {"messages": [{"role": "user", "content": "Weather in Portland?"}]}
)
print(result["messages"][-1].content)
```

Switching providers means changing only the model string, to ”openai:gpt-5” for example. The plain Python function becomes a tool, and its docstring, the comment in triple quotes under its first line, tells the model what the tool does.

Middleware lets you customize the loop without rewriting it. Each piece hooks in before or after the model call or a tool call. The prebuilt catalog includes these:

- Human-in-the-loop: pause before a risky tool call (sending email, writing to a database) so a person can approve, edit or reject it. Since v1.4.6 (October 2026), an opt-in mode gives each call its own approval prompt.
- Summarization: compress older messages when the conversation nears the model’s context limit, the most text it can read at once.
- PII redaction: strip personally identifiable information (PII), such as emails and phone numbers, before it reaches the model.
- Retries and fallbacks: retry a failed model call or switch to a backup model.
- Call limits: cap how many model or tool calls one run may make, which puts a ceiling on cost.

```
from langchain.agents.middleware import (
    HumanInTheLoopMiddleware, SummarizationMiddleware,
)

agent = create_agent(
    model="anthropic:claude-sonnet-4-5",
    tools=[send_email, search_docs],
    middleware=[
        SummarizationMiddleware(model="openai:gpt-5-mini"),
        HumanInTheLoopMiddleware(interrupt_on={"send_email": True}),
    ],
)
```

Those few lines put a person in front of every outgoing email and keep a long support thread inside the model's limit by having a cheaper model summarize the older messages while the agent carries on with the newer ones.

LangChain carries a few other everyday pieces:

- Structured output: ask for a Pydantic schema, a Python class that declares each field and its type, and get a typed object back. LangChain produces it inside the main loop, which saves an extra model call.
- Standard content blocks: .content_blocks has the same shape for reasoning, citations and tool calls, whichever provider answered.
- Retrieval-augmented generation (RAG): loaders, text splitters, embeddings (numeric fingerprints of meaning) and vector-store integrations let an agent search your own documents before it answers.
- MCP: since v1.4.0 (September 2026), the beta langchain.mcp package connects agents to the tools of any Model Context Protocol (MCP) server, the open standard for exposing tools to a model.

The older APIs moved out. Legacy chains, the old retrievers and the indexing helpers now live in a separate langchain-classic package, and version 1.x needs Python 3.10 or newer.

How LangGraph works

In LangGraph you draw the workflow yourself. Four ideas cover almost everything:

1. State is a typed object, such as a list of messages plus a few fields, that every step reads and updates.
2. Nodes are ordinary functions. One might call an LLM, another run a tool, another apply a hard-coded business rule.
3. Edges decide what runs next. A normal edge always goes to the same node. A conditional edge picks the next node from the state, and that's how you get branches and loops.

4. The checkpointer saves the state after every step to memory, SQLite or PostgreSQL. This one piece gives you pause and resume, crash recovery, multi-day runs and "time travel", which rewinds to an earlier step to try again.

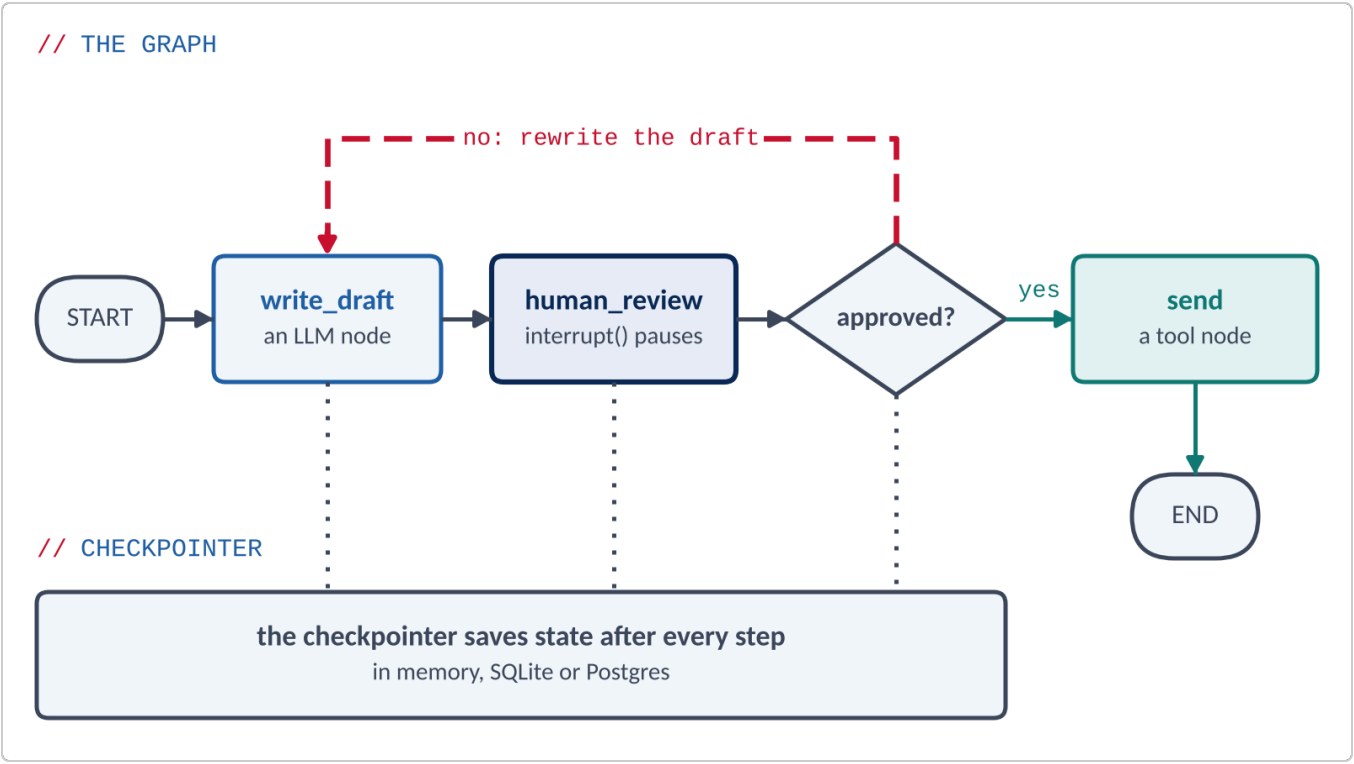


Figure 2. The run stops at human_review, and the saved checkpoint lets it resume there hours or days later.

The code below builds that workflow. A rejected draft loops back for a rewrite, and because the state is saved, the review can wait as long as it needs to.

```
from typing import TypedDict
from langgraph.graph import StateGraph, START, END
from langgraph.checkpoint.memory import InMemorySaver
from langgraph.types import interrupt

class State(TypedDict):
    request: str
    draft: str
    approved: bool

def write_draft(state: State):
    return {"draft": llm.invoke(f"Draft a reply:
```

```
{state['request']}").content}

def human_review(state: State):
    answer = interrupt({"draft": state["draft"]})  # pauses here
    return {"approved": answer == "yes"}

def route(state: State):
    return "send" if state["approved"] else "write_draft"

def send(state: State):
    email_client.send(state["draft"])
    return {}

g = StateGraph(State)
g.add_node("write_draft", write_draft)
g.add_node("human_review", human_review)
g.add_node("send", send)
g.add_edge(START, "write_draft")
g.add_edge("write_draft", "human_review")
g.add_conditional_edges("human_review", route)
g.add_edge("send", END)

app = g.compile(checkpointer=InMemorySaver())
```

Each run gets a thread_id. When the graph reaches interrupt(), it saves its state and stops. Hours or days later your app resumes that thread with the reviewer's answer, Command(resume="yes"), and the graph picks up at the review step with the draft still in hand. In production, swap InMemorySaver for a Postgres checkpointer so the state survives a restart.

LangGraph also handles these:

- Streaming sends tokens, step updates and custom events to a UI as they happen.
- Memory comes in two kinds, short-term memory per thread and a long-term store shared across sessions.
- Subgraphs and multi-agent setups work because a node can be a whole graph, such as a supervisor that routes work to specialist agents.

- Runtime controls arrived in v1.2.0 (May 2026): per-node timeouts, node-level error handlers that route to a recovery step, and a graceful shutdown that saves a resumable checkpoint.
- Mixing with LangChain is direct, since an agent built with create_agent can be dropped in as one node of a bigger graph.

What each is best for

Start with LangChain. Move a workflow to LangGraph when you need something from the right-hand column of this table. Most production systems end up there for part of their work.

Table 1. LangChain and LangGraph side by side.

	LANGCHAIN	LANGGRAPH
Level	High-level: one call builds an agent	Low-level: you design the graph
Control flow	A fixed loop: model, tools, model	Any shape: branches, loops, parallel steps
Long-running work	Inherits LangGraph's persistence, with little to configure	Built for it: multi-day runs, crash recovery
Human approval	Middleware on tool calls	Pause anywhere with interrupt()
Learning curve	Hours	Days
Best for	Chatbots, support bots, RAG question answering, tool-using assistants, prototypes	Approval workflows, multi-agent systems, mixed rule-and-AI pipelines, background jobs

Pick LangChain when:

- Your task fits ”ask the model, let it use tools, return an answer”.
- You want to try several model providers without rewriting code.
- You need a working prototype this week.
- You’re building question answering over your own documents.

Pick LangGraph when:

- Some steps must be deterministic, such as validating an invoice or applying a pricing rule, and others are AI-driven.
- A person must approve, edit or reject work partway through, possibly days later.
- The process runs for a long time or has to survive server restarts.
- Several specialist agents hand work to each other.
- You need tight control of latency and cost, deciding exactly when the model gets called.

Four real-world shapes come up again and again:

- Customer support assistant: a LangChain agent carries search and ticket tools, with human-approval middleware before any refund.
- Research assistant: a LangGraph graph plans, searches in parallel and summarizes, then critiques its own draft and loops until the result is good enough. Deep Agents packages this pattern.
- Document processing: a LangGraph pipeline extracts fields with an LLM, checks them with code and sends the exceptions to a person.
- Coding or operations agent: Deep Agents runs it, with sandboxed code execution and sub-agents.

Who uses them, their limits, and the alternatives

LangChain reported 90 million monthly downloads when the two libraries reached 1.0 in October 2025. The same announcement named Uber, J.P. Morgan, BlackRock and

Cisco as running production applications on the frameworks, and Uber, LinkedIn and Klarna as adopters of LangGraph.

A few things need watching.

- Fast release pace: between November 2025 and September 2026 LangChain shipped four minor releases, 1.1 through 1.4, and LangGraph shipped 1.1 and 1.2. Several headline features, among them MCP support, DeltaChannel checkpoints (which store only what changed) and v3 streaming, are still in beta. Pin your versions and read the changelog before upgrading.
- Abstraction overhead: for a single prompt-and-answer call, calling the model provider's own SDK (software development kit) directly is simpler.
- LangGraph's learning curve: thinking in state, nodes and edges takes time, and checkpoint storage needs a real database in production.
- Python leads JavaScript: new features usually ship in Python first.
- LangSmith costs money: the libraries are free under the MIT licence, and the hosted services on top of them are paid. The deployment service once called LangGraph Platform is now LangSmith Deployment.

Several alternatives are worth knowing: CrewAI (role-based multi-agent teams), Mastra (TypeScript-first agents), the Vercel AI SDK (chat interfaces in Next.js), LlamaIndex (document-heavy RAG), and the vendor SDKs, such as the OpenAI Agents SDK and the Claude Agent SDK, when you use a single model provider.

What we evaluated for MAX

MAX is an AI Resident: a personal AI that lives on one PC, listens, speaks, learns from what he reads and from how people respond, and keeps a record of everything he does. MAX4 is the Windows desktop app. MAX5 and MARA are the same resident at home and in a hospital. We looked at LangChain and LangGraph twice. The first look came while designing MAX3, the earlier web app. The second came on November 1, 2026, when the question was whether either library would help MAX4, MAX5 or MARA now.

The frameworks got a fair hearing. Here's what they would have given us:

- A large ecosystem: the libraries bring model adapters, vector stores, document loaders and tool wrappers that MAX otherwise writes himself.
- Proven agent patterns: the planning loops and tool routing are hard-won and well tested, and so are the retries around them.
- Durable state for free: LangGraph's checkpointer gives pause and resume, crash recovery and replay with almost no code.
- Human approval as a first-class feature: interrupt() and the human-in-the-loop middleware are exactly the kind of control MAX needs.
- Faster prototyping: any new agent feature would reach a working version sooner.

Those are real benefits. For MAX, the costs outweighed them.

Why we decided against them

1. The voice turn has a hard time budget. MAX is voice-first. From the moment someone stops speaking to the moment MAX starts answering, the whole turn must fit in about 1.5 to 2 seconds. That budget covers listening, retrieval, the model and speech. Every layer on that path has to justify its milliseconds, and a framework adds layers we'd have to measure and defend. Our design rule is that the person who owns the critical path must be able to read every line of it cold. A large, fast-moving dependency in the middle of the voice loop breaks that rule.

2. Most of MAX's work isn't an agent at all. MAX sorts every task into three tiers and always uses the lowest one that works:

- Tier 1, a direct call: the path is fully known: fetch a fact, save a record, capture a frame. It takes 50 to 200 ms, a tenth of the voice budget at most, and uses no tokens.
- Tier 2, a fixed workflow with LLM steps: the path is known, and some steps need judgment, such as classifying intent or drafting a reply. It takes 300 to 1,500 ms.
- Tier 3, an autonomous agent: the goal is known and the path isn't. It takes 2 to 15 seconds — on its own enough to overrun the whole voice turn — and it's less reliable.

The test is simple: if you can draw the flowchart before it runs, it isn't Tier 3. Almost everything MAX does is Tier 1 or Tier 2, and there a framework is pure overhead. The costliest mistake in agent design is using an agent where a plain call would do.

3. MAX's brain is Rust, in one process. MAX4 is a single Rust program, a Tauri 2 shell (a framework for desktop apps with a web front end and a Rust core) with the learner running inside it. No Python program stays resident. Python runs only as a one-off job, for a library with no Rust equivalent. It never writes the database. LangChain and LangGraph are Python and TypeScript libraries, so adopting them would mean a resident Python process and a hop between processes on every turn. The single-process rule exists to prevent exactly that.

4. One database, one writer. MAX keeps his memory as a graph in DuckDB, a database that runs inside the app's own process. It's opened once, and the Rust app is its only writer. LangGraph's checkpointer would bring a second store and a second writer for state, and MAX's architecture rules out a second graph writer.

5. MARA's audit rules want every step in our own code. Every action MAX takes is written to a hash-chained audit log, where each entry carries a fingerprint of the one before it, so a changed or missing entry breaks the chain. The entry lands in the same step as the change it records: who, what, why and how to undo it. The chain is checked when MAX starts, and MARA keeps it for six years. A framework's internal steps and retries would happen outside that chain. In a hospital, "the library did it" won't satisfy anyone reading the record.

6. Churn is a liability for a signed desktop app. The release pace above applies here: four minor LangChain releases in under a year, with headline features still in beta. MAX ships as a checked, signed installer and has a medical sibling. Pinning a version and hoping the next upgrade goes well is a plan nobody can defend to a clinic.

7. Spending and privacy are product rules. Swapping providers by changing a model string is convenient, but in MAX the choice of model is a decision with rules attached. The local model goes first. A paid cloud call waits for the Collective's approval and must fit the monthly budget. Every answer is audited. Those rules belong in code the Collective owns, where a middleware stack can't reorder them.

The same reasoning rules out CrewAI, AutoGen and Hugging Face's smolagents, which belong to the same class of framework.

What we chose instead

MAX uses a thin orchestration layer the Collective wrote and owns, roughly 500 to 1,000 lines on the critical path, which is small enough for one engineer to read cold. It runs inside the same Rust process as everything else. The table maps each thing LangChain or LangGraph would have given us to how MAX4 does it.

Table 2. What the frameworks offer, and how MAX4 does the same job.

WHAT YOU NEED	THE LANGCHAIN OR LANGGRAPH WAY	MAX4'S WAY
The agent loop	create_agent's loop: model, tools, model	One loop: retrieve first, call a model only on a miss, use the answer at its level of certainty, then learn from the person's response. Three tiers, lowest first
Switching model providers	Change the model string	Four small adapters in Rust: Anthropic's Messages API; one OpenAI-style adapter for OpenAI, xAI and the local Ollama, a tool that runs open models on the PC itself; Google's GenAI; and one that runs a provider's own command-line tool. Each model can be switched on or off by itself
Which model runs	Whatever the string says	MAX4's own local Ollama first, kept loaded and streaming. A cloud model is used only when the local one falls short, and only with the Collective's approval
Tools	A tools list, plus langchain.mcp	MCP, allowlisted. Every tool call counts as network traffic: an unknown host is refused, and a result is kept only if it can be cited
Human in the loop	Middleware, or interrupt()	A spend gate: a paid call waits until the Collective answers, with no timer. Autonomous actions must be reversible and come with a 30-second override. A skill below the "Trusted" stage waits for a person before it acts
Persistence and recovery	A checkpointer in memory, SQLite or Postgres	One DuckDB connection that Rust owns. A small supervisor reopens MAX4 2, 4 or 8 seconds after a crash
Memory and RAG	Loaders, splitters, vector stores	The graph in DuckDB, with Qdrant, a vector database, on the local machine ranking the vectors. Retrieval comes before any model, and a new fact needs a citable source
Cost limits	Call-limit middleware	The spend gate, plus a monthly budget kept by Muther, the house agent. A call over budget is held

WHAT YOU NEED	THE LANGCHAIN OR LANGGRAPH WAY	MAX4'S WAY
		even after a yes
Tracing	LangSmith, paid and hosted	The hash-chained audit log, local and checked at every start

As of November 2, 2026, MAX4's key storage, the supervisor that restarts him after a crash and the audit table are built. The provider adapters, the cloud calls, the audit chain's writer and the skill stages are designed and waiting to be built.

What we borrowed

Three of LangGraph's ideas are worth taking anyway, and MAX is taking them as patterns in Rust:

1. Explicit state graphs: LangGraph makes you draw the workflow, and so does MAX's tier rule. If a task's flowchart can be drawn, MAX draws it and writes it as plain code.
2. Save state after every step: a checkpointer after each node is what makes a LangGraph run resumable. MAX gets the same property by writing each change and its audit row in a single step, so he can always pick up from what's recorded.
3. Pause for a person: interrupt() is the cleanest version of human-in-the-loop we've seen. MAX's spend gate and override window apply the same idea where it matters most for him, which is money and actions in the real world.

One door stayed open when we reviewed MAX3. If a genuinely open-ended research agent at Tier 3 ever needs elaborate multi-step planning, a narrow use of a framework for that subsystem alone could be judged on its merits. It would sit behind a clean interface, away from the voice path. Nothing in MAX4 has needed it.

Why it matters

A framework decides some things for you — which model gets called, where state is kept, when a person is asked, what gets logged — and it decides them the way its authors needed most often. For a server-side or cloud application in Python or TypeScript, those defaults are good ones, and the 1.0 releases made them steadier. LangChain and LangGraph are a sound choice there, especially when durable approval workflows are needed quickly.

MAX's case runs the other way because each of those decisions is a rule with a consequence. Which model gets called is a question of money and privacy. Where state is kept is a question of who may write the memory. What gets logged is what a hospital auditor will read six years from now. Before adopting any agent framework, list the decisions it would make on your behalf, and mark the ones your product can't hand over. If the list comes back empty, take the framework and ship this week. If it doesn't, those lines are the code worth writing yourself, and the framework's best ideas can still come along as patterns.

Sources

- LangChain and LangGraph Agent Frameworks Reach v1.0 Milestones, LangChain blog, October 22, 2025.
  - langchain.com · langchain langgraph 1dot0
- LangGraph overview, LangChain docs.
  - docs.langchain.com · langgraph
- LangChain changelog, LangChain docs, read November 2, 2026.
  - docs.langchain.com · changelog.md
- LangChain 1.0 vs LangGraph 1.0: Which One to Use in 2026, ClickIT.
  - clickittech.com · langchain 1 0 vs langgraph 1 0
- LangChain and LangGraph in Late 2026: What Is New and Who Should Use It, Developers Digest.
  - developersdigest.tech · langchain langgraph whats new 2026

The code examples are simplified illustrations, so check the current docs for exact imports and model names.

Glossary

TERM	DEFINITION
Agent	An agent is a program in which a language model chooses its own next step, such as which tool to call, until it reaches an answer. MAX keeps agents for Tier 3, where the path can't be drawn in advance.
Audit log, hash-chained	A hash-chained audit log is a record of every action in which each entry carries a fingerprint of the entry before it, so a changed or missing entry breaks the chain. MAX checks the chain at every start, and MARA keeps it for six years.
Checkpointer	The checkpointer is the LangGraph component that saves a graph's state after every step, in memory, SQLite or PostgreSQL. It's what lets a run pause, recover from a crash or rewind.
Conditional edge	A conditional edge is an edge in a LangGraph graph that reads the state to choose the next node. Branches and loops are built from it.
create_agent	create_agent is LangChain's one-call builder for an agent from a model, a list of tools and a system prompt. It compiles into a LangGraph graph.
Deep Agents	Deep Agents is an opinionated agent package built on LangChain and LangGraph for long-running agents that plan, use files and start sub-agents.
DuckDB	DuckDB is a database that runs inside an application's own process rather than as a separate server. MAX keeps his memory graph in it, with the Rust app as the only writer.
Edge	An edge is an arrow in a LangGraph graph that decides which node runs next.
interrupt()	interrupt() is the LangGraph call that saves a run's state and stops until a person answers. The run then resumes at the same step.

TERM	DEFINITION
LangSmith	LangSmith is LangChain's paid platform for tracing, evaluating and deploying applications. It's optional, and its deployment service was formerly called LangGraph Platform.
Large language model (LLM)	A large language model is a model trained on large amounts of text that reads and writes language, such as the models behind ChatGPT and Claude. Both libraries exist to put one to work with tools and data.
Middleware	Middleware is code that hooks in before or after a model call or a tool call in a LangChain agent, to add approval, summarization, redaction, retries or limits without rewriting the loop.
Model Context Protocol (MCP)	The Model Context Protocol is an open standard for exposing tools and data to a model. MAX uses it with an allowlist, and LangChain added beta support in v1.4.0.
Node	A node is a box in a LangGraph graph: an ordinary function that calls a model, runs a tool or applies a rule.
Ollama	Ollama is a tool that runs open models on a local machine. MAX4 calls his local Ollama first and a cloud model only with approval.
Personally identifiable information (PII)	PII covers details such as names, emails and phone numbers that identify a person. LangChain's redaction middleware strips it before text reaches a model.
Pydantic	Pydantic is a Python library for declaring data classes with typed fields. LangChain uses a Pydantic schema to return structured output.
Qdrant	Qdrant is a vector database. MAX runs it on the local machine to rank vectors during retrieval.
Retrieval-augmented generation (RAG)	RAG means searching your own documents first and handing the results to the model with the question, so the answer rests on your sources.
Rust	Rust is a compiled programming language. MAX4 is a single Rust program, which is why Python libraries would need a second process.

TERM	DEFINITION
SDK (software development kit)	An SDK is a vendor's own library for calling its service, such as the OpenAI Agents SDK or the Claude Agent SDK.
Spend gate	The spend gate is MAX's rule that a paid cloud call waits for the Collective's yes and must fit the monthly budget. It plays the part a human-in-the-loop interrupt plays in LangGraph.
State	The state is the shared, typed object every step of a LangGraph graph reads and updates.
Tauri 2	Tauri 2 is a framework for desktop apps with a web front end and a Rust core. MAX4's shell is built on it.
Tier rule	The tier rule is MAX's practice of sorting every task into a direct call, a fixed workflow with model steps, or an autonomous agent, and always using the lowest tier that works.
Vector store	A vector store is a database that stores embeddings, numeric fingerprints of meaning, and finds the ones closest to a query. RAG uses it to find relevant passages.

AGENTIC AI WPR-AGT-009

// ABOUT THE AUTHOR

Jason Newell leads organizations and people in technology, and is a researcher, deep technologist and builder — from a CRAY to agentic AI. He founded MAX Research Collective to build AI for regulated industries that runs where its owner has control, with healthcare as its strongest focus. About Jason → Connect on LinkedIn →

Newell, J. (2026). MAX Research Collective, WPR-AGT-009. CC BY 4.0