LangChain — Components Deep Dive

A study guide to building agents with LangChain 1.x (Python), one component at a time: what it is, the code you will actually type, when to use it (and when not to), the traps, and where the official docs live.

> **How to read this guide.** Every component has (1) a mental model, usually an everyday analogy, (2) the code you will write, (3) a *when to use / when not to* verdict, (4) the traps, and (5) links to the official docs. The decision tables for **"when do I use tools?"** (§3) and **"when do I use memory?"** (§6) are the parts to re-read before a design review. The companion page is *LangGraph — Components Deep Dive*. **Version note:** written 17 Oct 2026 from docs.langchain.com. Features marked ⚑ need a minimum version (table in §14), and model names move fast, so run `pip show langchain langgraph` and pick current model IDs from your provider before copying anything.

---

0. The map: what LangChain is in 2026

> **Analogy: a car. LangChain** is the cockpit: steering wheel, pedals and dashboard that feel the same in every car (any model provider). **LangGraph** is the engine and chassis: it keeps the car running, lets you stop and restart a long trip, and gives you low-level control. **Deep Agents** is a fully equipped rental: GPS, spare tyre and a trunk (planning, subagents, filesystem) already installed.

	Runtime	Framework	Harness

Example	**LangGraph** (also Temporal, Inngest)	**LangChain** (also Vercel AI SDK, CrewAI, OpenAI Agents SDK, Google ADK, LlamaIndex)	**Deep Agents SDK** (also Claude Agent SDK)
Value add	Durable execution, streaming, human-in-the-loop, persistence	Abstractions and integrations	Predefined tools, prompts, subagents, planning, filesystem
Use when	Low-level control, long-running stateful workflows, mixing deterministic and agentic steps	Getting started quickly, standardising how a team builds agents	Autonomous agents on complex, non-deterministic, multi-step tasks

Packages. `langchain` (agents, messages, tools, embeddings), `langchain-core` (base abstractions), one package per provider (install by extra: `pip install 'langchain[anthropic]'`), `langgraph` (runtime, checkpointers, stores), and `langchain-classic` (legacy chains, retrievers, hub and community re-exports, moved out of the main namespace in v1).

What changed in v1 (the things old tutorials get wrong):

- `create_agent` is *the* way to build an agent. The old `prompt` parameter is now `system_prompt`; dynamic prompts go through middleware.
- **Middleware** is the defining feature: hooks around the model call and tool call replace most custom graph surgery.
- Messages expose **standard content blocks** (`content_blocks`) that look the same across providers.
- The `langchain` namespace is deliberately small. If `from langchain.chains import ...` or `from langchain import hub` fails, you want `pip install langchain-classic`.
- Agent state schemas must be `TypedDict` (no Pydantic state in `create_agent`).

Choosing the layer (quick verdict): single model call such as classify, extract or summarise → a bare model (§1). Model plus tools in a loop → `create_agent`. Need branching, parallel fan-out, custom loops or durable multi-step flows → LangGraph (companion page), and you can still embed a `create_agent` agent as a node. Long autonomous work with files and subagents → Deep Agents.

Runtimes, frameworks, harnesses · What's new in v1 · Migration guide · Philosophy

---

1. Models

> **Analogy: an engine you can swap.** The model is the reasoning engine. LangChain gives every engine the same plug (invoke, stream, batch, bind_tools, structured output), so you can change provider without rewriting the car.

1.1 Initialise and call

```
from langchain.chat_models import init_chat_model

model = init_chat_model('anthropic:claude-sonnet-5', temperature=0.2, timeout=30, max_tokens=1000, max_retries=6)

model.invoke('Why do parrots talk?')            # one AIMessage
for chunk in model.stream('Tell me a story'):    # AIMessageChunk objects, sum them to get the full message
    print(chunk.text, end='', flush=True)
model.batch(['q1', 'q2', 'q3'], config={'max_concurrency': 5})   # client-side parallelism
```

- **Two ways in:** `init_chat_model('provider:model')`, or the provider class (`ChatAnthropic`, `ChatOpenAI`...). Model names are passed straight to the provider API, so a new model works without a LangChain update.
- **Resilience is built in:** `max_retries` defaults to **6** with exponential backoff and jitter for network errors, 429 and 5xx. 401 and 404 are not retried. For long agent runs on shaky networks raise it to 10-15 and add a checkpointer.
- **`batch()` is not the provider's batch API.** It parallelises calls client-side; provider batch endpoints are a separate product.
- **Configurable models:** `init_chat_model(temperature=0)` with no model name lets you pick the model per call via `config={'configurable': {'model': '...'}}`. Use `config_prefix` when a chain has several models.

1.2 Tool calling and structured output on a bare model

```
model_with_tools = model.bind_tools([get_weather])            # tool_choice='any' forces a call
ai = model_with_tools.invoke('Weather in Boston and Tokyo?')    # ai.tool_calls may hold several parallel calls

from pydantic import BaseModel, Field
class Movie(BaseModel):
    title: str = Field(description='Title')
    year: int
model.with_structured_output(Movie).invoke('Details on Inception')   # Movie(...)
```

With a bare model **you** run the tool and send a `ToolMessage` back. An agent (§4) does that loop for you. `with_structured_output` accepts Pydantic, `TypedDict` or JSON Schema; `method` is `json_schema` (provider-native), `function_calling` (forces a tool call) or `json_mode` (valid JSON, schema only in the

prompt). `include_raw=True` returns the raw message too, which is how you get token counts.

1.3 Other model features worth knowing

Feature	What to know
Reasoning	Read `content_blocks` of type `reasoning`. ⚑ A standard `reasoning_effort` parameter exists for several providers; check `model.profile['reasoning_effort_levels']`.
Model profile ⚑	`model.profile` lists capabilities (`max_input_tokens`, `tool_calling`, `structured_output`, modalities). Summarisation middleware and structured-output auto-selection read it. It is beta and can be overridden with `profile=`.
Multimodal	Pass `{'type': 'image', 'url': ...}` or base64 plus `mime_type` as content blocks; not every model supports every type.
Prompt caching	Three levels: implicit (OpenAI, Gemini), explicit provider controls (`cache_control` blocks for Anthropic, `cachePoint` for Bedrock), and middleware that caches stable system prompt and tool content. Usually only engages above a minimum token count.
Server-side tools	Web search and code interpreters run at the provider. The response carries `server_tool_call` and `server_tool_result` blocks in one turn, with *no* ToolMessage for you to return.
Rate limiting	`rate_limiter=InMemoryRateLimiter(requests_per_second=...)` limits request count only, not request size.
Exceptions	Standard types (`ModelRateLimitError`, `ModelTimeoutError`, `ContextOverflowError`...) each carry `is_retryable`, which retry middleware respects.
Token usage	`msg.usage_metadata`, or aggregate across models with `get_usage_metadata_callback()`.

When to use a bare model: one-shot generation, classification, extraction, routing decisions inside your own code, or evaluator calls. **When not to:** as soon as the model must choose tools and loop, use an agent.

Traps. (1) `model.invoke` returning a *string* means you built a legacy text-completion LLM, not a chat model. (2) A pre-bound model (`bind_tools` already called) cannot be used where `create_agent` needs structured output. (3) `parallel_tool_calls=False` exists on some providers if your tools are not safe to run concurrently.

□□ Models · Providers and models · Chat model integrations

---

2. Messages

> **Analogy: a transcript with stage directions.** Each line has a speaker (role), the words (content) and margin notes (metadata such as IDs and token counts). The whole conversation is just a list of these lines.

Type	Role	Notes
`SystemMessage`	Instructions that prime behaviour	Always a SystemMessage inside `ModelRequest.system_message`, even when you passed a string.
`HumanMessage`	User input, text or multimodal	A plain string is shorthand for one HumanMessage.
`AIMessage`	Model output	`.text`, `.content`, `.content_blocks`, `.tool_calls`, `.usage_metadata`, `.response_metadata`, `.id`.
`ToolMessage`	Result of one tool call	`tool_call_id` must match the AIMessage's call ID. `artifact` carries data for your app that is *not* sent to the model.

Standard content blocks (`msg.content_blocks`): `text` (with annotations/citations), `reasoning`, `image`, `audio`, `video`, `file`, `text-plain`, `tool_call`, `tool_call_chunk`, `invalid_tool_call`, `server_tool_call`, `server_tool_result` and the escape hatch `non_standard`. `content` stays loosely typed (string, provider-native blocks or standard blocks); `content_blocks` lazily parses it into the standard form. Set `output_version='v1'` (or `LC_OUTPUT_VERSION=v1`) if another system must read the standard blocks from stored content.

Rules that bite. Every tool call in history needs a matching ToolMessage, and most providers want history to start with a user message. Streaming chunks (`AIMessageChunk`) add up with `+`. Serialisation uses `dumpd` and `load`, but **never call `load()` on untrusted data**: it instantiates Python objects and can trigger side effects.

Messages

---

3. Tools

>

> **Analogy: hands for a brain.** The model can only think and talk. A tool lets it reach out: read a database, call an API, write a file. The model decides *whether* and *with what arguments* to reach; your code does the reaching.

3.1 Define a tool

```
from langchain.tools import tool


@tool
def search_orders(user_id: str, status: str, limit: int = 10) -> str:
    '''Search a user's orders by status. Use when the user asks about order history.

    Args:
        user_id: Unique user ID
        status: pending, shipped or delivered
        limit: Max results
    '''
    ...
```

- **Type hints are required** (they become the input schema). The docstring is the description the model reads, so write it as a prompt: when to use it, what the arguments mean.
- Prefer `snake_case` names; spaces and special characters get rejected by some providers. Override with `@tool('web_search')` or `description=...`.
- Richer inputs: `@tool(args_schema=PydanticModel)` or a JSON schema.
- **Reserved argument names:** `config` and `runtime`. Do not use them for your own parameters.

3.2 ToolRuntime: state, context, store, streaming

Add `runtime: ToolRuntime` to the signature. It is injected at call time and *hidden from the model's schema*.

Field	What it is	Use for
`runtime.state`	Short-term memory for this thread (messages, custom fields)	Read history, counters, auth flags

Field	What it is	Use for
`runtime.context`	Immutable per-run config passed at invoke time	User ID, tenant, API keys, permissions
`runtime.store`	Long-term memory, persists across threads	Preferences, learned facts
`runtime.stream_writer`	Emit custom progress events	'Fetched 10/100 records'
`runtime.tool_call_id`	ID of this call	Building a matching ToolMessage
`runtime.execution_info` ⚑	thread_id, run_id, attempt number	Logging, retry-aware behaviour
`runtime.server_info` ⚑	Assistant/graph ID, authenticated user on LangGraph Server (else None)	Per-user scoping on the platform

```
from langchain.tools import tool, ToolRuntime
from langchain.messages import ToolMessage
from langgraph.types import Command

@tool
def set_user_name(new_name: str, runtime: ToolRuntime) -> Command:
    '''Save the user's name.'''
    return Command(update={
        'user_name': new_name,
        'messages': [ToolMessage(content='Name saved.', tool_call_id=runtime.tool_call_id)],
    })
```

3.3 What a tool may return

Return	Effect	Pick it when
`str`	Becomes a ToolMessage the model reads	Result is naturally text
`dict`/object	Serialised and returned; the model reads fields	Downstream reasoning needs explicit fields
Content-block list	Multimodal ToolMessage (text plus image...)	Screenshots, charts; model must support the modality
`Command(update=..., goto=...)`	Writes to state and can route	The tool must change state. **Include a ToolMessage with the matching `tool_call_id`** or ToolNode raises `ValueError`.
`@tool(return_direct=True)`	Skips the final model call and returns the tool output	Output is already the complete answer

Parallel gotchas. Models may call several tools in one step. If two tools write the same state key, define a *reducer* for it. `return_direct` only short-circuits the loop when *every* tool call in that step has it set; a mixed batch goes back to the model.

3.4 Errors, dynamic tools, headless tools, MCP

- **Errors:** use `ToolErrorMiddleware` ⚑ or a `@wrap_tool_call` that catches the exception and returns a `ToolMessage` so the model can self-correct. Return the exception *type* rather than the raw message when details could be sensitive. Put retry *inner* (later in the list) so exceptions reach the error handler after retries are exhausted.
- **Dynamic tool selection:** if all tools are known up front, register them all and filter per call with `@wrap_model_call` and `request.override(tools=...)` (by auth state, role, feature flag). If tools are discovered at runtime (MCP, registry), you need *two* hooks: `wrap_model_call` to add them and `wrap_tool_call` to execute them, otherwise the agent cannot run the tool it just advertised.
- **Headless tools:** schema-only tools whose implementation lives in the client (browser geolocation, clipboard). The run interrupts with a tool payload and resumes when the client returns a result.
- **MCP ⚑ (beta):** `from langchain.mcp import MCPAdapter`; `async with MCPAdapter(url) as a: tools = await a.list_tools()`, then pass to `create_agent`. Requires `langchain[mcp]>=1.4.0`; the standalone `langchain-mcp-adapters` package has a migration guide.

3.5 WHEN to use tools (decision table)

Situation	Use	Why
Needs fresh or private data (DB, API, files, web)	**Tool**	The model cannot know it
Needs exact arithmetic, code execution, or a guaranteed-correct lookup	**Tool**	Models are unreliable calculators
Has a side effect (send, write, delete, pay)	**Tool plus human approval (§9)**	Gate the dangerous ones
Step is always executed, in a fixed order, with no judgement needed	**Plain code or a graph node**, not a tool	Do not pay tokens and latency to have a model 'decide' the obvious
You only need the final answer in a fixed shape	**Structured output (§8)**	Cheaper and safer than a tool that just returns data
Static knowledge that fits in the prompt	**Prompt or skill**	No call needed
Big knowledge base	**Retrieval tool (§12)**	Fetch only relevant chunks

Situation	Use	Why
More than roughly 10 tools and the model picks wrong	**Filter dynamically, LLMToolSelector, skills or subagents**	Too many tools overload context and raise errors
Tool output is huge	**Context editing or filesystem middleware**	Stops results filling the window

Traps. Vague docstrings cause wrong calls. A tool that returns a 50k-token blob poisons every later turn. A tool that mutates state without returning a ToolMessage breaks message history. Tools run again if a node is retried or resumed, so make side effects **idempotent**.

Tools · MCP · Headless tools · Prebuilt tools

---

4. Agents (create_agent)

> **Analogy: a new hire with a toolbox and a handbook. Agent = model + harness.** The model does the thinking; the harness (prompt, tools, middleware) decides what it sees and may do. The loop is: think, pick a tool, see the result, think again, until it answers.

```
from langchain.agents import create_agent
from langgraph.checkpoint.memory import InMemorySaver
from langchain_core.utils.uuid import uuid7

agent = create_agent(
    model='anthropic:claude-sonnet-5',
    tools=[search_orders],
    system_prompt='You are a concise support agent.',
    checkpointer=InMemorySaver(),          # dev only
    name='support_agent',
)
cfg = {'configurable': {'thread_id': str(uuid7())}}
r1 = agent.invoke({'messages': [{'role': 'user', 'content': 'Where is order 42?'}]}, config=cfg)
r2 = agent.invoke({'messages': [{'role': 'user', 'content': 'And the one before?'}]}, config=cfg)   # same thread, remembers
```

Parameter	Purpose

`model`	'provider:model' string or model instance (not pre-bound if you use structured output)
`tools`	Callables, `@tool` objects or provider tool dicts
`system_prompt`	String or `SystemMessage` (use a SystemMessage for cache control)
`middleware`	The customisation primitive (§5)
`response_format`	Schema for the final structured answer (§8)
`state_schema`	Extra state fields (TypedDict only). Prefer defining them on the middleware that uses them
`context_schema`	Shape of per-run immutable context (`runtime.context`)
`checkpointer`	Thread persistence (short-term memory, HITL, time travel)
`store`	Cross-thread long-term memory
`name`	Identifier; vital when the agent is a subgraph, and surfaces as `lc_agent_name` in stream metadata

thread_id versus context. `thread_id` scopes the *conversation* (history and checkpoints). `context=` carries *per-run* data (user ID, keys). In production pass both on every call.

Result shape. `result['messages']` is the full history; `result['structured_response']` exists when `response_format` is set. With `version='v2'`, invoke returns an object with `.value` and `.interrupts` (⚑ LangGraph 1.1+).

Limits. `recursion_limit` is a top-level key of `config` (not inside `configurable`); LangGraph 1.0.6+ defaults it to 1000 steps and raises `GraphRecursionError` beyond it. Add `ModelCallLimitMiddleware` so a looping agent cannot burn money.

When create_agent is enough: a model with tools, memory, guardrails and approvals. **When to drop to LangGraph:** explicit branching and fan-out you must control, deterministic steps interleaved with agentic ones, or custom loops.

□□ Agents · Quickstart · Runtime

---

5. Middleware

> **Analogy: airport security checkpoints along the agent's route.** Before boarding (before_agent), before each flight (before_model), wrapped around the flight itself (wrap_model_call), after landing (after_model), around each baggage

> handoff (wrap_tool_call), and at the exit (after_agent). Each checkpoint can inspect, change or turn the passenger around.

Style	Hook	Runs
Node	`before_agent` / `after_agent`	Once per invocation
Node	`before_model` / `after_model`	Around each model response
Wrap	`wrap_model_call`	Around each model call (you choose how often to call `handler`: 0 = short-circuit, 1 = normal, N = retry)
Wrap	`wrap_tool_call`	Around each tool call
Convenience	`@dynamic_prompt`	Build the system prompt per call

Order. `before_*` run first to last; `after_*` run last to first; `wrap_*` nest like function calls (first middleware is outermost). Put critical guards first.

```
from langchain.agents.middleware import before_model, wrap_model_call, dynamic_prompt, AgentState,
ModelRequest, ModelResponse
from langgraph.runtime import Runtime
from langchain.messages import AIMessage

@before_model(can_jump_to=['end'])
def stop_long_chats(state: AgentState, runtime: Runtime):
    if len(state['messages']) >= 50:
        return {'messages': [AIMessage('Conversation limit reached.')], 'jump_to': 'end'}

@dynamic_prompt
def role_prompt(request: ModelRequest) -> str:
    return f'You are a helpful assistant for a {request.runtime.context.user_role}.'

@wrap_model_call
def route_model(request: ModelRequest, handler) -> ModelResponse:
    model = big_model if len(request.messages) > 10 else small_model
    return handler(request.override(model=model))
```

- **Jumps:** return `{'jump_to': 'end' | 'tools' | 'model'}`, and declare the target with `can_jump_to` or `@hook_config`.
- **State updates:** node hooks return a dict applied through the graph's reducers. Wrap hooks return `ExtendedModelResponse(model_response=..., command=Command(update=...))` for model calls, or a `Command` for tool calls. With several layers, commands compose: messages are additive, and for non-reducer keys the *outermost* value wins. If an outer

layer retries, commands from earlier attempts are dropped.

- **Middleware-scoped state:** subclass `AgentMiddleware`, set `state_schema = MyState` (extra fields live and die with that middleware), and optionally ship `tools` (for example `write_todos`) and stream `transformers` ⚑.
- **Class versus decorator:** decorators for one hook; classes for several hooks, configuration, or when you need both sync and async (`abefore_model`...).
- **Middleware is not a separate runtime.** The hooks run inside the compiled LangGraph, so you can embed the whole agent as a node in a larger `StateGraph` and its HITL, summarisation and retries come with it.

5.1 Built-in middleware catalogue

Middleware	Use it for	Key options
`ToolErrorMiddleware` ⚑	Turn tool exceptions into error ToolMessages the model can recover from	`on_error` (return None to propagate), `tools`
`ToolRetryMiddleware`	Retry flaky tools with backoff	`max_retries=2`, `backoff_factor=2.0`, `retry_on`, `on_failure='continue'|'error'`
`ModelRetryMiddleware`	Retry model calls	Same options; ⚑ default respects `is_retryable`
`ModelFallbackMiddleware`	Fail over to other models/providers	Ordered list of fallback models
`SummarizationMiddleware`	Compress old history *persistently*	`trigger` (tokens/messages/fraction, AND within a dict, OR across a list), `keep`
`ContextEditingMiddleware`	Clear old tool outputs when context grows	`ClearToolUsesEdit(trigger, keep=3, exclude_tools)`
`HumanInTheLoopMiddleware`	Approve, edit, reject tool calls	`interrupt_on`, `interrupt_mode`, `when` ⚑
`ModelCallLimitMiddleware`	Cap model calls (cost, loops)	`thread_limit` (needs checkpointer), `run_limit`, `exit_behavior`
`ToolCallLimitMiddleware`	Cap total or per-tool calls	`tool_name`, `thread_limit`, `run_limit`, `exit_behavior`
`PIIMiddleware`	Detect/handle PII	strategy `block|redact|mask|hash`, custom `detector`, `apply_to_input/output/tool_results`
`TodoListMiddleware`	Planning via a `write_todos` tool	Custom prompt/description
`LLMToolSelectorMiddleware`	Pre-select relevant tools when you have 10+	`model`, `max_tools`, `always_include`
`ProviderToolSearchMiddleware`	Defer tools behind provider-side tool search	`searchable_tools`; Anthropic and recent OpenAI models only
`ShellToolMiddleware`	Persistent shell session	Execution policy: host, Docker or sandbox. Not compatible with interrupts yet

Middleware	Use it for	Key options
`FilesystemFileSearchMiddleware`	Glob and grep tools over a directory	`root_path`, `use_ripgrep`
`FilesystemMiddleware`, `SubAgentMiddleware`, `MemoryMiddleware`, `SkillsMiddleware` (deepagents)	Files as context, delegation, memory files, on-demand skills	Backends: `StateBackend`, `StoreBackend`, `CompositeBackend`
`RubricMiddleware` (beta)	Self-grade against a rubric and iterate	`max_iterations`
`LLMToolEmulator`	Fake tool results for tests	`tools`, `model`
Provider-specific	Anthropic prompt caching, bash, text editor, memory; Bedrock prompt caching; OpenAI moderation	See integrations

A sensible production stack (outer to inner): call limits → HITL → PII → summarisation or context editing → model retry/fallback → tool error → tool retry. The one ordering rule worth memorising: **HITL must wrap retry and error middleware** in per-call mode, or a retried call asks the reviewer again.

Traps. Keep middleware single-purpose. Hooks that run before a HITL pause run *again* on resume, so they must be side-effect free. `thread_limit` on any limiter silently does nothing without a checkpointer. Summarisation compresses text only; images are not shrunk.

Overview · Prebuilt · Custom

---

6. Memory: when to use which

> **Analogy: a notepad on your desk versus a filing cabinet in the archive.** The notepad (short-term memory) holds *this* conversation and is thrown away or archived with it. The cabinet (long-term memory) holds things worth knowing in *any* future conversation. The desk is small, so you also need a rule for when to summarise the notepad.

6.1 Decision table

You need…	Use	Mechanism
The agent to remember earlier turns of this conversation	**Short-term memory**	`checkpointer` plus a stable `thread_id`

You need…	Use	Mechanism
Pause and resume later (approvals, crashes)	**Short-term memory**	Same checkpointer; required for HITL
The conversation is getting too long for the context window	**Trim, delete or summarise**	`@before_model` trim, `RemoveMessage`, or `SummarizationMiddleware`
Remember a user across sessions (preferences, profile, learned facts)	**Long-term memory**	`store` with namespace and key; read/write from tools or middleware
Find memories by meaning, not exact key	**Store with semantic index**	`InMemoryStore(index={'embed':..., 'dims':...})` then `store.search(query=...)`
Per-request settings (user ID, tenant, keys)	**Not memory: runtime context**	`context_schema` plus `context=` on invoke
Large intermediate artefacts (search dumps, files)	**Filesystem**, not messages	`FilesystemMiddleware`; route `/memories/` to a `StoreBackend` to persist
Static reference knowledge	**Not memory: prompt, skills or retrieval**	§12

6.2 Short-term memory (thread scope)

```
from langgraph.checkpoint.postgres import PostgresSaver      # pip install langgraph-checkpoint-postgres 'psycopg[binary]'
with PostgresSaver.from_conn_string(DB_URI) as cp:
    cp.setup()                                               # creates tables
    agent = create_agent('openai:gpt-5.5', tools=[...], checkpointer=cp)
    agent.invoke({'messages': 'Hi, I am Bob'}, {'configurable': {'thread_id': 'cust-17'}})
```

- State is read at the start of each step and written when a step completes. Custom fields: subclass `AgentState` (TypedDict) and pass it through `state_schema` or, better, a middleware's `state_schema`.
- `InMemorySaver` is for development; use Postgres, SQLite, Mongo or Cosmos-backed savers in production.

```
# Trim (transient is possible via wrap_model_call; this one rewrites state)
from langchain.messages import RemoveMessage
from langgraph.graph.message import REMOVE_ALL_MESSAGES
@before_model
def trim(state: AgentState, runtime: Runtime):
    m = state['messages']
    if len(m) <= 3: return None
    return {'messages': [RemoveMessage(id=REMOVE_ALL_MESSAGES), m[0], *m[-3:]]}
```

```
# Summarise (persistent)
SummarizationMiddleware(model='openai:gpt-5.4-mini', trigger=('tokens', 4000), keep=('messages', 20))
```

Choosing a strategy. *Trim* is cheap and lossy: fine for chit-chat. *Delete* is surgical: remove specific messages. *Summarise* costs an extra model call but keeps meaning: use for long support or research threads. *Context editing* clears old tool outputs only: use when tool results are what bloat the window. **Always keep history valid**: a tool-call AIMessage must be followed by its ToolMessage, and some providers want a user message first. `RemoveMessage` only works on a key with the `add_messages` reducer (the default `messages` key has it).

6.3 Long-term memory (cross-thread)

```
from langgraph.store.memory import InMemoryStore            # PostgresStore in production
store = InMemoryStore(index={'embed': embeddings, 'dims': 1536, 'fields': ['$']})
store.put(('user_123', 'memories'), 'm1', {'text': 'Prefers short, direct answers'})
hits = store.search(('user_123', 'memories'), query='communication style', limit=3)

@tool
def save_memory(text: str, runtime: ToolRuntime[Context]) -> str:
    '''Save a durable fact about the user.'''
    runtime.store.put((runtime.context.user_id, 'memories'), str(uuid7()), {'text': text})
    return 'saved'

agent = create_agent(model, tools=[save_memory], store=store, context_schema=Context)
```

- **Model:** JSON documents under a `namespace` tuple (like a folder, usually including user or org ID) and a `key` (like a file name). Cross-namespace search is by prefix and content filter.
- **Nothing is injected automatically.** You must *read* memories (a tool the model can call, or a `@dynamic_prompt`/`wrap_model_call` that injects them) and *write* them (a tool, or a background step after the turn). Decide deliberately between 'model decides what to remember' (tool, flexible, sometimes forgets) and 'we extract after every turn' (middleware, consistent, costs a call).
- Use user-scoped namespaces so one user can never read another's memories. Namespace prefix matching returns sub-namespaces too.
- On the managed platform the store is provisioned for you; semantic search needs index settings in `langgraph.json`.

Traps. (1) Reusing a `thread_id` across users leaks history. (2) A fresh `thread_id` means a blank notepad, not a lost cabinet: long-term memory survives because it is in the store. (3) `InMemoryStore` and `InMemorySaver` vanish on restart. (4) Summarisation permanently replaces messages; if you need the originals for audit, write them elsewhere first.

7. Context engineering

> **Analogy: packing the briefing folder for a consultant.** The consultant is capable, but if the folder holds the wrong pages, the advice will be wrong. Most agent failures are the wrong folder, not a weak consultant.

Agents usually fail because the right context was not passed to the model, not because the model is too weak. LangChain splits what you control into three kinds:

Context type	You control	Lifetime
Model context	System prompt, messages, tools, model, response format for one call	**Transient** (does not change saved state)
Tool context	What tools read and write (state, store, runtime context)	Persistent
Life-cycle context	What happens between steps: summarise, guardrails, logging	Persistent

Data source	Also called	Scope	Examples
Runtime context	Static config	One run	User ID, role, API keys, environment
State	Short-term memory	One conversation	Messages, uploaded files, auth status
Store	Long-term memory	Across conversations	Preferences, extracted insights

- **Transient versus persistent.** `wrap_model_call` with `request.override(...)` changes only what this call sees. To change saved history, return a `Command` from the wrap hook or use `before_model`, `after_model` or a tool.
- **Recency matters.** Models pay more attention to the last messages, so injected context (compliance rules, style guide) is appended *after* history rather than before it.
- **Typical moves:** dynamic system prompt by role or environment; inject uploaded-file summaries; filter tools by authentication state; route to a bigger model for long threads; switch `response_format` by stage.
- **Start simple:** static prompt and tools first, add one dynamic behaviour at a time, and watch token use and latency.

8. Structured output

> **Analogy: a form instead of an essay.** You hand the model a form with typed boxes. You get back something your code can use without parsing prose.

```
from pydantic import BaseModel, Field
from langchain.agents.structured_output import ToolStrategy, ProviderStrategy

class Ticket(BaseModel):
    category: str = Field(description='billing, technical, account or product')
    priority: str
    summary: str

agent = create_agent(model='openai:gpt-5.5', tools=[], response_format=Ticket)   # auto strategy
r = agent.invoke({'messages': [{'role': 'user', 'content': 'My card was charged twice!'}]})
r['structured_response']      # Ticket(...)
```

Option	Behaviour	Pick it when
Pass a schema type	Auto: provider-native if the model supports it, else tool-calling	Default
`ProviderStrategy(schema, strict=...)` ⚑	Provider enforces the schema; most reliable	Provider supports it; need strict adherence
`ToolStrategy(schema, handle_errors=...)`	Schema becomes a tool; model must call it; validation errors are fed back for retry	Provider lacks native support; union types

- **Schema types:** Pydantic (returns validated instance), dataclass or TypedDict (return a dict), JSON Schema (needs top-level `title` and `description`, and **must be wrapped** in an explicit strategy), and `Union[...]` with ToolStrategy so the model picks one shape.
- **`handle_errors`:** `True` (default, retry with a template message), a string (always that message), an exception type or tuple (only those), a callable returning a message, or `False` (raise). If the model returns two structured outputs at once, the agent feeds back an error and asks for one.
- `tool_message_content` customises the confirmation line that lands in history.
- If you pass tools *and* a schema, the model must support using both together.

Structured output versus a tool: use `response_format` for the *final answer's shape*; use a tool when something must *happen* mid-run. Outside agents, use `model.with_structured_output` (§1).

📖 Structured output

---

9. Human-in-the-loop and guardrails

> **Analogy: a second signature on the cheque.** The agent may prepare the transfer, but for large amounts a human signs before it goes out. The agent waits, with its place saved, as long as it takes.

```
from langchain.agents.middleware import HumanInTheLoopMiddleware
from langgraph.types import Command

agent = create_agent('openai:gpt-5.5', tools=[write_file, execute_sql, read_data],
    middleware=[HumanInTheLoopMiddleware(interrupt_on={
        'write_file': True,                          # all decisions allowed
        'execute_sql': {'allowed_decisions': ['approve', 'reject']},
        'read_data': False,                          # auto-approve
    })],
    checkpointer=InMemorySaver())                    # REQUIRED
cfg = {'configurable': {'thread_id': 't1'}}
r = agent.invoke({'messages': [...]}, config=cfg, version='v2')
r.interrupts                                         # what needs review
agent.invoke(Command(resume={'decisions': [{'type': 'approve'}]}), config=cfg, version='v2')   # same thread_id
```

Decision	Effect	Note
`approve`	Run as proposed	
`edit`	Run with changed args (`edited_action`)	Edit conservatively; big edits make the model re-plan and may repeat the call
`reject`	Skip and send feedback message	Use this to deny. Say whether the agent may retry

Decision	Effect	Note
`respond`	Your reply becomes the tool result	For 'ask user' tools only. The model sees a *success*, so never use it to deny

- **Batched mode (default):** one interrupt per model turn listing all gated calls; resume with an ordered `decisions` list. **`interrupt_mode='per_call'` ⚑:** one interrupt per call, each with a `response_schema`, resumed with a dict keyed by interrupt ID. The docs recommend per-call for new agents and plan to make it the default next major version.
- **Conditional gating ⚑:** `'when': predicate(request)` pauses only risky calls (for example SQL that is not a SELECT). The predicate re-runs on resume and must give the same answer, or the reviewer's decision is ignored.
- In per-call mode an invalid answer is rejected before saving, so send a corrected answer to the same interrupt ID. If several answers are sent together and one is invalid, others may have run, so validate first or answer one at a time.
- Stream while pausing: `stream = agent.stream_events(..., version='v3')`, check `stream.interrupted` and `stream.interrupts`.

Guardrails are policies enforced deterministically regardless of what the model does: `PIIMiddleware` (redact/mask/hash/block, with custom detectors), call limits, and your own `after_agent` checks (including model-based safety checks using `jump_to='end'`). A system prompt asking the model to behave is not a guardrail.

□□ Human-in-the-loop · HITL concepts · Guardrails

---

10. Streaming

> **Analogy: a live sports ticker instead of the final score.** You can show progress (each step), words as they are typed (tokens), or your own custom events ('*Fetched 10/100 records*').

Mode	Yields
`updates`	State update after each agent step (model, tools, model)
`messages`	`(token, metadata)` from any node that calls an LLM (metadata has `langgraph_node`, `lc_agent_name`)
`custom`	Anything written with `runtime.stream_writer` or `get_stream_writer()`
`values`	Full state after each step

```
# Classic modes: v2 format = one dict per chunk with type, ns, data
for chunk in agent.stream(inp, stream_mode=['updates', 'messages', 'custom'], version='v2'):
    if chunk['type'] == 'messages':
        token, meta = chunk['data']

# Event streaming (recommended for new apps, langchain 1.3+): one typed iterator per projection
stream = agent.stream_events(inp, config=cfg, version='v3')
for kind, item in stream.interleave('messages', 'tool_calls'):
    if kind == 'messages':
        for t in item.text: print(t, end='')
final = stream.output
```

- **Reasoning tokens:** iterate `message.reasoning` in event streaming, or filter `content_blocks` for `reasoning`.
- **Subagents and nested graphs:** pass `subgraphs=True` on classic streaming, or read `stream.subgraphs` in event streaming. Name every agent so you can tell who is talking.
- **Tool streaming:** tool calls arrive as `tool_call_chunk` blocks (partial JSON); complete calls come from `updates` or by summing chunks until `chunk_position == 'last'`.
- **Turn it off** for a model with `streaming=False` (or `disable_streaming=True`) when you do not want its tokens reaching the client.
- If a custom tool uses `get_stream_writer`, it can no longer run outside a LangGraph context.

Streaming · Event streaming · Frontend

---

11. Multi-agent patterns

> **Analogy: a hospital.** A generalist (single agent) can handle most cases. When the case needs a specialist's long reference books, you either call the specialist in (**subagent**), transfer the patient (**handoff**), look up a protocol (**skill**), or triage at the door (**router**).

First question: do you need multi-agent at all? A single agent with the right tools and prompt often suffices. Reach for multi-agent when (a) context management: specialised knowledge would overflow one prompt, (b) distributed development: separate teams own capabilities, or (c) parallelisation. Also when one agent has too many tools and picks poorly.

Pattern	How it works	Strengths

Subagents	Main agent calls subagents as tools; all routing goes through it	Isolation, parallel calls, large-context domains; stateless by default
Handoffs	Tool calls change a state variable; the active agent or its prompt/tools switch	Direct user conversation with the specialist; stateful
Skills	One agent loads specialised prompts on demand	Simple, directly conversational; context accumulates
Router	A step classifies input and sends it to one or more agents, then synthesises	Parallel fan-out across knowledge sources; stateless
Custom workflow	LangGraph graph mixing deterministic and agentic nodes	Full control; embed any other pattern as a node

Measured trade-offs from the docs: one-shot 'buy coffee' costs 4 model calls with subagents but 3 with handoffs, skills or router. A repeat request costs 8 total with subagents (stateless) versus 5 with handoffs or skills (stateful). Multi-domain comparison (three languages, about 2K tokens of docs each): subagents and router about 9K tokens, skills about 15K (context accumulates), handoffs 14K+ and sequential. Rule of thumb: **parallel and big-context → subagents or router; repeat and conversational → handoffs or skills.**

```
@tool
def ask_fruit_expert(question: str) -> str:
    '''Ask the fruit expert. Use for ALL fruit questions.'''
    r = fruit_agent.invoke({'messages': [{'role': 'user', 'content': question}]})
    return r['messages'][-1].content
supervisor = create_agent(model, tools=[ask_fruit_expert, ask_veggie_expert], checkpointer=cp)
```

Traps. A per-thread stateful subagent (`checkpointer=True`) breaks when the model calls it twice in parallel (checkpoint conflict): add `ToolCallLimitMiddleware(tool_name=..., run_limit=1)`. The `langgraph-supervisor` package has a migration guide to this tool-wrapped pattern. Deep Agents ships subagents, skills, planning and filesystem if you would rather not assemble this yourself.

Multi-agent · Subagents · Handoffs · Skills · Router · Custom workflow

---

12. Retrieval (RAG)

> **Analogy: an open-book exam.** Instead of memorising the library, the model gets a librarian (the retriever) who fetches the right pages when asked.

Scope note: I did not re-read the RAG tutorials for this page; this section is the stable shape plus pointers. Verify names against the tutorials. The pipeline is: load documents → split into chunks → embed (`init_embeddings`) → store in a vector store → retrieve. Two ways to use it: **agentic RAG** (expose a retrieval *tool*; the model decides when to search and can search several times) and **two-step RAG** (always retrieve first, then answer; predictable and cheap). Return the passage as the ToolMessage `content` and put document IDs or page numbers in `artifact` so your UI can cite without bloating the prompt.

Use RAG when knowledge is large, changes often, or must be cited. **Do not** use it for facts that fit in the prompt, or as a substitute for structured data you can query exactly (use a SQL tool). **Memory versus RAG:** memory is about *this user and these conversations*; RAG is about *shared documents*.

Build a semantic search engine · Agentic RAG with LangGraph · SQL agent · Vector stores

---

13. Production: reliability, cost, security, testing

- **Reliability stack:** model `max_retries` plus `ModelRetryMiddleware`, `ModelFallbackMiddleware` for provider outages, tool retry and error middleware, a durable checkpointer, and call limits as circuit breakers.
- **Cost:** prompt caching, summarisation or context editing, a cheap model for routing and summaries (a `wrap_model_call` router), tool filtering, and `ModelCallLimitMiddleware`.
- **Security:** treat tool output as untrusted input (prompt injection can arrive inside a web page or document); gate side effects with HITL; run shell tools under Docker or a sandbox policy, never the host policy on untrusted input; redaction rules do not stop exfiltration on the host; use PII middleware; sanitise error text returned to the model; never `load()` untrusted serialised messages; scope store namespaces per user.
- **Observability:** set up LangSmith tracing from day one; it records every turn, tool call and middleware span. Use a `trace_policy` ⚑ to omit large payloads.
- **Testing:** unit-test logic with fake chat models and in-memory persistence (no API calls); integration-test with real models (manage keys, flakiness, cost); evaluate trajectories with deterministic matching or LLM-as-judge; `LLMToolEmulator` fakes tools when real ones are expensive.
- **Deployment:** LangSmith deployment provisions checkpointer and store for you; locally pass them explicitly.

Testing · Observability · Deployment · LangSmith Studio

---

14. Version gates and the trap list

⚑ Feature	Minimum
Model profiles, fraction triggers in summarisation	langchain 1.1
`ProviderStrategy(strict=...)`	langchain 1.2
Middleware stream transformers; PII output redaction in streams; event streaming	langchain 1.3.x (1.3.2 for transformers/PII)
HITL `when` predicate	1.3.3
`ToolErrorMiddleware`	1.3.14
Middleware `trace_policy`	1.3.15
Retry middleware respects `is_retryable` by default	1.3.16
`langchain.mcp` (beta)	`langchain[mcp]` 1.4.0
HITL `interrupt_mode='per_call'`	1.4.6
`reasoning_effort` standard parameter	langchain-core 1.5.2 plus provider package versions
`runtime.execution_info`, `server_info`; v2 stream format	langgraph 1.1.x

Trap checklist (read before every review):

1. No checkpointer, yet expecting memory or HITL. Or a reused `thread_id` across users.
2. `InMemory*` in production.
3. Tool with a vague docstring, or no ToolMessage after returning a Command.
4. Too many tools; huge tool outputs; no call limits.
5. Retry placed outside HITL; side effects before an interrupt.
6. Structured output with a pre-bound model, or a raw JSON Schema not wrapped in a strategy.
7. Old tutorial imports (`prompt=`, `langchain.chains`, Pydantic agent state) on a v1 install.
8. Trusting the system prompt as a security control.

---

15. Exercises

1. Build an agent with one `@tool` that reads `runtime.context.user_id`, a `checkpointer`, and a `thread_id`. Prove it remembers turn 1 on turn 2, then change only the `thread_id` and show it forgets.
2. Add `SummarizationMiddleware` with a tiny trigger (`('messages', 6)`), run 10 turns, and print the history to see the summary message replace the old ones.
3. Add a `save_memory` tool plus an `InMemoryStore`. In thread A tell the agent a preference; in thread B (new `thread_id`, same `user_id`) show it can recall it via a search tool.

4. Gate `execute_sql` with HITL so only non-SELECT queries pause (a `when` predicate). Approve one, reject one with a message, and observe how the model reacts to the rejection.
5. Write a `wrap_model_call` router that uses a cheap model for the first 6 messages and a stronger one after. Verify with `usage_metadata` or LangSmith.

Solution notes

1. Same `thread_id` reloads state from the checkpointer; a new one starts empty. `context` is separate and must be passed on every call.
2. Summarisation is persistent: the checkpointed state now holds a summary message plus the last `keep` messages. Trim, by contrast, only changes what is sent if done in `wrap_model_call`.
3. The store key space is `(user_id, 'memories')`, independent of thread. The agent will not use the memory unless a tool or prompt injection reads it.
4. `when` returns True to pause. On resume it is evaluated again and must match. A reject message should tell the agent whether to retry.
5. `request.override(model=...)` changes only this call; models should be created once outside the hook.

Sources: docs.langchain.com (Python): LangChain overview, products, agents, models, messages, tools, short-term memory, long-term memory, middleware, structured output, HITL, streaming, multi-agent, context engineering. RAG section: scope note above.

Revision #1
Created 17 October 2026 21:10:57 by N8N
Updated 17 October 2026 21:10:57 by N8N