

Building an AI Agent

From First Principles to a Working Build

What an agent actually is, how the loop works, and how to build one

Eamon Wyse

Wyse Production

A technical primer · Concepts and hands-on build

About This Document

This is a primer for anyone who wants to understand what an AI agent actually is and how to build one — a developer, a technically-minded producer, or a curious reader who can follow a little code. No prior experience with agents is assumed. It comes in two parts: first the concepts, in plain language and with no code, so the ideas land before any syntax does; then a hands-on build you can actually run.

The build uses Python and the Anthropic API directly, with no framework. That choice is deliberate. Frameworks hide the very thing this document is trying to teach — the loop at the heart of every agent. Working against the raw API means nothing is concealed: you see the whole mechanism with your own eyes, and once you understand it, every framework reveals itself as a convenience wrapper around exactly what you will have built here. It is also the most durable approach, since the underlying concepts outlast any particular library.

What you'll need to run the build (Part 2):

- **Python 3** installed on your machine.
- **The Anthropic SDK:** `pip install anthropic`.
- **An Anthropic API key**, set as an environment variable (`ANTHROPIC_API_KEY`) rather than written into your code. The first build step shows how.

A note on specifics. The concepts here are evergreen, but exact details — the API model name, the precise SDK call — can change over time. Where this document shows a specific model string or function, treat it as current-at-writing and check Anthropic's documentation if something has moved. The **loop**, the four ingredients, and the way tool use works will not change.

Part 1 — What an Agent Is

The one-sentence definition

An AI agent is a language model placed inside a loop, given the ability to call tools, and allowed to keep going until a task is done. That is the whole idea. Everything else is detail. A plain language model answers a single question and stops. An agent is the same model wired into a cycle where it can take actions in the world, see the results, and decide what to do next — repeatedly, on its own, until the goal is met.

It helps to be precise about what is and is not new here. The **intelligence** lives entirely in the model. The **agency** — the capacity to act over multiple steps — comes from the loop and the tools you build around it. You are not making the model smarter. You are giving it hands, and a reason to use them more than once.

Chatbot vs. agent: the actual difference

People blur these two together, but the distinction is sharp and worth holding onto:

A plain chatbot	An agent
Receives a message, returns a reply, stops.	Receives a goal, then works toward it over many steps.
Cannot affect anything outside the conversation.	Can call tools that read files, search the web, run code, hit APIs.
Has no memory of actions — there are none.	Observes the result of each action and adapts the next one.
You drive every turn.	It drives its own turns until it decides it is finished.

The crucial word in the right-hand column is **decides**. An agent is not following a fixed script you wrote. At each step it looks at everything that has happened so far and chooses the next action itself. Your job as the builder is not to choreograph the steps — it is to define the goal, supply the tools, and run the loop.

The four ingredients

Every agent, however sophisticated, is assembled from exactly four parts. If you can name these four, you understand the architecture:

1. **The model.** The reasoning engine — a large language model such as Claude. It does the thinking: interpreting the goal, choosing actions, interpreting results.
2. **The tools.** Functions the model is allowed to call — search the web, read a file, query a database, send an email. Each tool is something the agent can do that the model alone cannot.
3. **The loop.** The control structure that runs the model, executes whatever tool it asked for, feeds the result back, and runs the model again. This is the beating heart of agency.

4. **The context.** The running transcript of everything so far — the goal, every action taken, every result returned. This is the agent's working memory; it is what the model re-reads on every pass through the loop.

Hold these four in mind and the rest of this document is just showing you how they fit together. The model and tools are components you configure. The loop and the context are things **you** build and manage. That last point surprises people: the model does not remember anything on its own. Memory is something you construct by accumulating the transcript and handing it back each time.

How tool use actually works

This is the mechanism that makes agents possible, and it is more mundane than it sounds. The model cannot **actually** run code or read a file. What it can do is **ask** — it emits a structured message that says, in effect, “I would like to call the tool named `read_file` with the argument `path = report.txt`.” That request is just text in a defined format.

Your program — not the model — sees that request, runs the real function, and hands the result back into the conversation. The model then continues as if it had always known the answer. So the division of labour is strict:

- **The model decides** which tool to call and with what arguments.
- **Your code executes** the tool and returns the result.
- **The model interprets** the result and decides what to do next.

The model never touches your files, your network, or your database directly. It only ever asks. You remain in complete control of what actually runs — which is exactly where safety and trust come from. We will return to that.

The agent loop, in plain words

Before any code, here is the entire control flow of an agent described as a recipe. Read it once and the program later will feel inevitable:

5. Give the model the goal and the list of tools it may use.
6. Ask the model what to do next.
7. The model responds in one of two ways: either it gives a **final answer** (the task is done), or it **requests a tool call**.
8. If it is a final answer — **stop**. Return it to the user.
9. If it is a tool call — run that tool, capture the result, append both the request and the result to the context, and **go back to step 2**.

That is the whole thing. A loop that keeps asking the model “what now?”, doing whatever it asks, and showing it the outcome — until the model says it is finished. Everything labelled “agent”, from a toy script to a system that books your travel, is a variation on these five steps.

Part 2 — Building One

We will now build a real, working agent using the Anthropic API directly in Python, with no framework. The deliberate choice to avoid frameworks is so that nothing is hidden: you will see the loop with your own eyes. Once you understand this, every framework reveals itself as a convenience wrapper around exactly what follows.

What we are building

A small research assistant agent. You give it a question; it can use two tools — a calculator and a (mock) web search — and it works through the problem on its own, calling tools as needed, until it can answer. Two tools is enough to demonstrate every principle; going from two to twenty is repetition, not new concepts.

Step 0 — Setup

Install the SDK and set your API key as an environment variable. Never hard-code a key into source you will share or commit.

```
pip install anthropic

# In your shell (Mac/Linux):
export ANTHROPIC_API_KEY="sk-ant-..."
```

Step 1 — Define the tools

A tool has two halves that must be kept in sync. First, a **description** the model reads — a name, an explanation of what the tool does, and a schema for its inputs. This is the only thing the model knows about the tool, so the wording matters: it is how the model decides **when** to reach for it. Second, the **actual function** your code runs when the model asks. Here are the descriptions:

```
tools = [
    {
        "name": "calculator",
        "description": "Evaluate a basic arithmetic expression and "
            "return the numeric result. Use this whenever the "
            "task involves a calculation.",
        "input_schema": {
            "type": "object",
            "properties": {
                "expression": {
                    "type": "string",
                    "description": "A math expression, e.g. '142 * 17'"
                }
            },
            "required": ["expression"]
        }
    },
    {
        "name": "web_search",
```

```

        "description": "Search the web for current information on a topic. "
        "Use this for facts you do not know or that may have "
        "changed recently.",
        "input_schema": {
            "type": "object",
            "properties": {
                "query": {
                    "type": "string",
                    "description": "The search query"
                }
            },
            "required": ["query"]
        }
    }
}

```

Now the real functions. The model never sees this code — it only ever sees the descriptions above. (The search here is faked so the example runs without external dependencies; swap in a real search API and nothing else changes.)

```

def calculator(expression):
    # In production, use a safe parser, not raw eval.
    return str(eval(expression))

def web_search(query):
    # Mocked for the example. Replace with a real API call.
    fake_results = {
        "speed of light": "299,792,458 metres per second",
        "population of france": "About 68 million (2024)",
    }
    for key, value in fake_results.items():
        if key in query.lower():
            return value
    return "No results found."

# A lookup so the loop can find the function by its tool name.
TOOL_FUNCTIONS = {
    "calculator": calculator,
    "web_search": web_search,
}

```

Step 2 — The agent loop

This is the centre of everything. Read it slowly; it is the five-step recipe from Part 1 turned into code. Notice that **messages** is the context — the growing transcript — and that we append to it on every pass so the model always sees the full history.

```

import anthropic

client = anthropic.Anthropic() # reads ANTHROPIC_API_KEY

def run_agent(user_goal):

```

```

# The context: the running transcript. Starts with the user's goal.
messages = [{"role": "user", "content": user_goal}]

while True: # the loop
    # --- Ask the model what to do next ---
    response = client.messages.create(
        model="claude-sonnet-4-6",
        max_tokens=1024,
        tools=tools,
        messages=messages,
    )

    # Record what the model said into the transcript.
    messages.append({"role": "assistant", "content": response.content})

    # --- Did it finish, or does it want a tool? ---
    if response.stop_reason != "tool_use":
        # No tool requested => this is the final answer. Stop.
        final = "".join(
            block.text for block in response.content
            if block.type == "text"
        )
        return final

    # --- It asked for one or more tools. Run them. ---
    tool_results = []
    for block in response.content:
        if block.type == "tool_use":
            fn = TOOL_FUNCTIONS[block.name] # find the real function
            result = fn(**block.input)      # run it with model's args
            tool_results.append({
                "type": "tool_result",
                "tool_use_id": block.id,    # ties result to request
                "content": result,
            })

    # Feed the results back in as the next user turn, then loop.
    messages.append({"role": "user", "content": tool_results})

```

Step 3 — Run it

```

answer = run_agent(
    "What is the speed of light in metres per second, "
    "and how far does light travel in 7 seconds?"
)
print(answer)

```

Watching what happens internally is the best way to feel how an agent thinks. The loop runs several times for this single question:

10. **Pass 1:** The model reads the goal, realises it does not know the speed of light for certain, and requests `web_search(query="speed of light")`. The loop runs it, returns "299,792,458 m/s".

11. **Pass 2:** The model now has the figure and needs the distance. It requests `calculator(expression="299792458 * 7")`. The loop runs it, returns the product.
12. **Pass 3:** The model has everything it needs. It returns a final text answer and no tool call, so `stop_reason` is not `tool_use` — the loop exits and hands the answer back.

Nobody told the agent to search first and calculate second. It **worked that out** — that ordering emerged from the model reasoning about the goal at each step. That emergent, self-directed sequencing is the whole point. You supplied capability and a loop; the strategy was the model's.

What you just built, restated

Step back and map the code onto the four ingredients from Part 1:

Ingredient	Where it lives in the code
The model	<code>client.messages.create(...)</code>
The tools	the <code>tools</code> list + the real Python functions
The loop	the <code>while True:</code> block
The context	the <code>messages</code> list, appended to each pass

That really is a complete agent. Everything beyond this is adding more or better tools, managing the context as it grows, and making it safe and reliable — not new architecture.

The Complete Program

The sections above introduced the agent in pieces, which is the right way to learn it. Here is the whole thing assembled into a single file you can save as `agent.py` and run. Nothing here is new — it is exactly the tools, functions, and loop you have already seen, plus the connective tissue a runnable program needs: the imports, the `MAX_STEPS` guard wired in, error handling around each tool call, and an entry point so it runs when executed. Read it as confirmation that the whole agent really does fit on a single page of code.

The code is also a companion file

The full listing is reproduced below so you can read it in place. The same code is also provided as a companion file, `agent.py`, ready to run — use that to avoid copying from the page, which can disturb Python's indentation. If you only have this document and not the file, the listing below is complete and self-contained: typing or pasting it as shown gives you exactly the same working program. Two further companion files, `agent_local.py` and `agent_live.py`, extend this same loop with real tools; they are described at the end of this part.

```
"""
agent.py – A minimal AI agent, built from first principles.
A model in a loop with tools: the model thinks, the tools act,
the loop persists, the context remembers.

Requires:  pip install anthropic
           export ANTHROPIC_API_KEY="sk-ant-..."
Run:       python agent.py
"""

import anthropic

client = anthropic.Anthropic() # reads ANTHROPIC_API_KEY from the environment

# A safety guard so a confused agent cannot loop forever.
MAX_STEPS = 10

# -----
# 1. TOOL DESCRIPTIONS – the only thing the model knows about the tools.
# The wording here is effectively a prompt: it decides when the model
# reaches for each tool.
# -----
tools = [
    {
        "name": "calculator",
        "description": "Evaluate a basic arithmetic expression and return "
                       "the numeric result. Use this whenever the task "
                       "involves a calculation.",
        "input_schema": {
            "type": "object",
            "properties": {
                "expression": {
```

```

        "type": "string",
        "description": "A math expression, e.g. '142 * 17'",
    },
    },
    "required": ["expression"],
},
{
    "name": "web_search",
    "description": "Search the web for current information on a topic. "
                  "Use this for facts you do not know or that may have "
                  "changed recently.",
    "input_schema": {
        "type": "object",
        "properties": {
            "query": {"type": "string", "description": "The search query"}
        },
        "required": ["query"],
    },
},
],

# -----
# 2. THE REAL FUNCTIONS – what your code actually runs. The model never
#   sees this code; it only ever sees the descriptions above.
# -----
def calculator(expression):
    # NOTE: eval() is used here ONLY to keep the example short. Never ship
    # eval() on untrusted input – use a real expression parser in production.
    return str(eval(expression))

def web_search(query):
    # Mocked so the example runs with zero setup. Replace the body with a
    # real search API call and nothing else in this file needs to change.
    fake_results = {
        "speed of light": "299,792,458 metres per second",
        "population of france": "About 68 million (2024)",
    }
    for key, value in fake_results.items():
        if key in query.lower():
            return value
    return "No results found."

# A lookup so the loop can find the real function by the tool's name.
TOOL_FUNCTIONS = {
    "calculator": calculator,
    "web_search": web_search,
}

# -----

```

```

# 3. THE AGENT LOOP – the heart of everything. 'messages' is the context:
#   the running transcript, appended to on every pass so the model always
#   sees the full history.
# -----
def run_agent(user_goal, verbose=True):
    messages = [{"role": "user", "content": user_goal}]

    for step in range(MAX_STEPS):
        response = client.messages.create(
            model="claude-sonnet-4-6",
            max_tokens=1024,
            tools=tools,
            messages=messages,
        )

        # Record what the model said into the transcript.
        messages.append({"role": "assistant", "content": response.content})

        # Did it finish, or does it want a tool?
        if response.stop_reason != "tool_use":
            return "".join(
                block.text for block in response.content
                if block.type == "text"
            )

        # It asked for one or more tools. Run each one.
        tool_results = []
        for block in response.content:
            if block.type == "tool_use":
                if verbose:
                    print(f" [step {step + 1}] calling "
                          f"{block.name}({block.input})")
                try:
                    fn = TOOL_FUNCTIONS[block.name]
                    result = fn(**block.input)
                except Exception as e:
                    # Hand the failure back to the model as the result;
                    # it can then adapt rather than crash.
                    result = f"Error running {block.name}: {e}"
                tool_results.append({
                    "type": "tool_result",
                    "tool_use_id": block.id,
                    "content": result,
                })

        # Feed the results back as the next turn, then loop.
        messages.append({"role": "user", "content": tool_results})

    return "Stopped: reached the maximum number of steps."

# -----
# 4. ENTRY POINT – runs only when you execute this file directly.
# -----

```

```
if __name__ == "__main__":
    goal = ("What is the speed of light in metres per second, "
           "and how far does light travel in 7 seconds?")
    print(f"Goal: {goal}\n")
    answer = run_agent(goal)
    print(f"\nAnswer: {answer}")
```

How to Run It

Three steps, assuming the setup from Step 0:

1. **Save** the listing above as `agent.py`.
2. **Set your key** in the shell: `export ANTHROPIC_API_KEY="sk-ant-..."` (and `pip install anthropic` if you have not already).
3. **Run it**: `python agent.py`.

You will see the agent print each tool call as it makes it, then the final answer — the loop working in front of you. Change the `goal` string at the bottom to ask it something else. To give it real abilities, replace the mocked `web_search` body with a call to an actual search API; because the loop only cares about the tool's **description** and its return value, nothing else in the file needs to change.

One honest caveat about this code

The `eval()` inside `calculator` and the mocked `web_search` are deliberate shortcuts to keep the example short and runnable with zero setup. `eval()` on untrusted input is genuinely unsafe — fine for learning on your own machine, never for anything real. Part 3's section on safety explains what to do instead. This is exactly the “validate inputs” point made concrete.

Going Further: Two Working Versions

The `agent.py` above is the teaching version: it runs on an Anthropic key alone, but its tools are deliberately mocked so the focus stays on the loop. Two further companion files take the same loop — unchanged — and give it tools that do real work. They form a progression: learn the mechanism, then run something genuinely useful, then reach the internet.

File	What it adds	Setup needed
<code>agent.py</code>	The teaching version. Mocked search, safe to read, runs anywhere.	Anthropic key only
<code>agent_local.py</code>	Real local tools: file read/write, directory listing, and a genuinely safe calculator (an arithmetic parser, not <code>eval</code>). Acts on a confined workspace folder.	Anthropic key only
<code>agent_live.py</code>	Everything in local, plus real web search. The search is provider-agnostic — point it at any search API via two environment variables, no code change.	Anthropic key + a search provider key

Why two extra files instead of one. The split is the point. `agent_local.py` runs with nothing but the key you already have, so anyone can clone it and watch a real agent work in under a minute — no accounts, no external services. `agent_live.py` then shows the next rung: integrating a real outside service cleanly, with the vendor isolated behind a single swappable function and the key handled through the environment rather than hard-coded.

The safe calculator, made concrete

The caveat above warned never to ship `eval()`. Here is what the companion files do instead — parse the expression into a syntax tree and walk it, permitting only arithmetic, so no arbitrary code can run:

```
import ast, operator

_ALLOWED_OPS = {
    ast.Add: operator.add, ast.Sub: operator.sub,
    ast.Mult: operator.mul, ast.Div: operator.truediv,
    ast.Pow: operator.pow, ast.USub: operator.neg,
}

def _safe_eval(node):
    if isinstance(node, ast.Constant) and isinstance(node.value, (int, float)):
        return node.value
    if isinstance(node, ast.BinOp) and type(node.op) in _ALLOWED_OPS:
        return _ALLOWED_OPS[type(node.op)](_safe_eval(node.left),
                                             _safe_eval(node.right))
    if isinstance(node, ast.UnaryOp) and type(node.op) in _ALLOWED_OPS:
        return _ALLOWED_OPS[type(node.op)](_safe_eval(node.operand))
    raise ValueError("Unsupported expression")

def calculator(expression):
    return str(_safe_eval(ast.parse(expression, mode="eval").body))
```

Legitimate maths works exactly as before; an input like `__import__('os').system(...)` is rejected rather than executed. The file tools apply the same discipline to the filesystem — every path is resolved and checked to be inside a single workspace folder, so the agent cannot read or overwrite anything elsewhere on the machine. This is the “least privilege” principle from Part 3, built in rather than bolted on.

Swapping the search provider

In `agent_live.py` the web search speaks to whatever JSON search API you configure through two environment variables — no vendor is named in the code:

```
export SEARCH_API_URL="https://api.your-provider.com/search"
export SEARCH_API_KEY="..."
```

If your provider's request or response shape differs from the common pattern, you adjust one function and nothing else. And if the variables are not set at all, the agent still runs: the search tool reports that it is unconfigured, and the model adapts and uses its other tools — the graceful-degradation idea from Part 3, made real.

Part 3 — From Toy to Real

The loop above is genuinely the core of production agents. But shipping one to real users surfaces a handful of concerns the toy version ignores. None changes the architecture; each is a layer of care on top of it.

The system prompt: giving the agent a brief

In the example the agent had only the user's goal to go on. Real agents are given a **system prompt** — standing instructions that frame who the agent is, what it should and should not do, how to use its tools, and when to stop. It is the difference between handing someone a task and handing someone a task plus a job description.

```
response = client.messages.create(
    model="claude-sonnet-4-6",
    max_tokens=1024,
    system=("You are a careful research assistant. Use tools "
           "when you are unsure of a fact. Show your reasoning. "
           "If a question is ambiguous, ask before acting."),
    tools=tools,
    messages=messages,
)
```

Knowing when to stop

Our loop trusts the model to stop by eventually not requesting a tool. Usually it does. But a confused agent can loop — calling tools forever, or repeating the same failing call. Production loops add a guard rail:

```
MAX_STEPS = 10
steps = 0
while steps < MAX_STEPS:
    steps += 1
    # ... rest of the loop ...
# If we fall out here, stop and report rather than spin forever.
```

A step cap is the simplest safeguard. More mature systems also watch for repeated identical calls and for runaway cost, since every pass through the loop is a paid model call.

Errors and the real world

Tools fail. APIs time out, a file is missing, a search returns nothing. The elegant part is that an agent handles this naturally if you let it: catch the error and **return it to the model as the tool result**. The model reads “that failed” and adapts — retries, tries another approach, or explains the problem to the user. The failure becomes just another observation in the loop.

```
try:
    result = fn(**block.input)
except Exception as e:
    result = f"Error running {block.name}: {e}"
```

```
# Either way, 'result' goes back to the model. It decides what to do.
```

Context: the memory problem

The messages list grows on every pass. For a short task this is fine. For a long-running agent it eventually exceeds the model's context window — the maximum amount of text it can read at once — and costs rise with every appended turn, since the whole transcript is re-sent each time. Real systems manage this by:

- **Summarising** older parts of the transcript once they are no longer needed in full.
- **Storing** results externally (a file, a database) and keeping only a reference in the context.
- **Pruning** tool results that are no longer relevant to the remaining work.

This is one of the genuinely hard parts of agent engineering, and most of the difference between a demo and a dependable system lives here, in how cleverly the context is curated.

Safety and trust

Recall the strict division of labour: the model only ever **asks** to run a tool; your code decides whether to actually run it. That boundary is your primary safety control, and you should treat it as one. Practical measures:

- **Least privilege.** Give the agent only the tools it needs. An agent that should read files does not also need one that deletes them.
- **Confirmation gates.** For consequential actions — sending money, emailing a client, deleting data — have your code pause and ask a human before executing, even though the model requested it.
- **Validate inputs.** The model's tool arguments are suggestions, not commands you must obey blindly. Check them. The `eval()` in our calculator is a deliberate example of what not to ship.
- **Sandbox.** If the agent runs code or touches a filesystem, contain it so a mistake cannot reach anything that matters.

The mindset that keeps you safe: the model is a capable, well-meaning, occasionally mistaken collaborator. You would not give a new collaborator unsupervised access to production on day one. The same caution, encoded in your tool layer, is how you build agents you can trust.

Where to go next

Once the loop is second nature, the natural directions to grow are:

- **More and richer tools** — real web search, code execution, database access, third-party APIs. Capability scales with tools.
- **Multi-agent systems** — several specialised agents that hand work to one another, with one orchestrating the rest.

- **Frameworks** — having built one by hand, libraries that manage loops, context, and tools become a convenience rather than a black box. You will know exactly what they are doing for you.
- **Evaluation** — systematically measuring whether your agent succeeds, so changes are improvements you can prove rather than hope for.

The one thing to remember

If you forget everything else, keep this: **an agent is a model in a loop with tools**. The model thinks, the tools act, the loop persists, the context remembers. Every agent you will ever meet — however large, however clever — is built from those four parts arranged the way you just saw. You now know how it all works.