Agents: Using and Extending Them

Agentic Cybersecurity — Module 2 (double module)

LLM + tools + loop = agent

What Is an Agent?

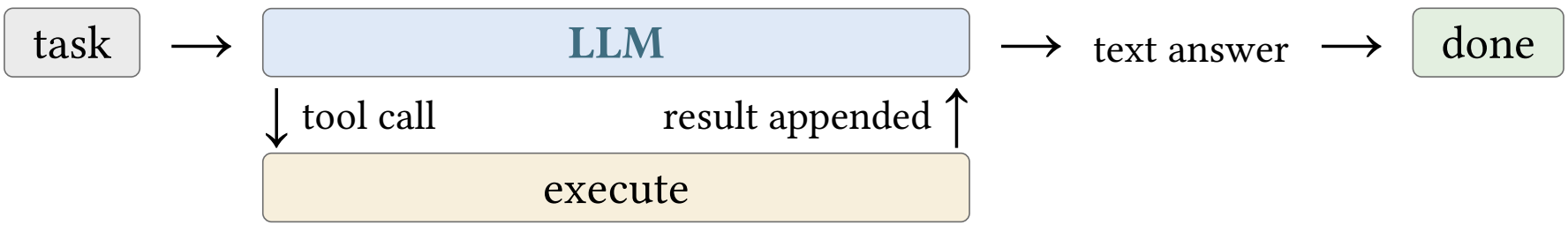
The Definition

An **agent** is an LLM given tools and run in a loop.

1. Model receives a task (and tool definitions)
2. Model responds with either an answer **or a tool call**
3. Harness executes the tool, appends the result to the context
4. Go to 2

That's it. Everything else — "autonomy", "reasoning", "planning" — is emergent behavior of this loop plus a good model.

The Loop, As a Picture



Two exits from the LLM: say something (done), or ask for something (another lap).

This picture is the whole module. Every slide from here on is pointing at one of these boxes.

Chatbot vs. Agent

Chatbot	Agent
talks about doing things	does things
you copy/paste results	it reads and writes directly
one shot, your context	iterates, builds its own context
errors end the exchange	errors get fed back and fixed

The loop is what lets an agent **recover**: run the tests, see the failure, fix, repeat. That feedback cycle is most of the value.

Tool Calling

How It Actually Works

The model "calls" a tool by emitting structured output.

- Request out: the conversation **plus** your tool definitions
- Response back: normal text, **or** a tool call
- The harness executes, appends the result, and calls again
- The model **never executes anything** — it only asks

You'll hand-build exactly this in Lab 2.1, atop your Lab 1.1 script.

Consequence: every tool result is **input the model will act on**. Remember that in Module 4.

Tool Calls Are Not Special

What a tool call looks like on the wire:

```
{"role": "assistant", "tool_calls": [{"id": "call_a1b2c3",
  "function": {"name": "read_file",
               "arguments": "{\"path\": \"src/auth.py\"}"}}]}
```

- The provider parses the model's **emitted tokens** into this shape
- Underneath: still next-token prediction that happens to parse

The model emits text; the harness acts on it.

How Does the Model Know Tools Exist?

They ride in the request: name, description, JSON schema.

```
{"type": "function", "function": {"name": "bash",
  "description": "Execute a bash command. Returns stdout and
stderr.",
  "parameters": {"type": "object", "properties":
    {"command": {"type": "string"}}, "required":
["command"]}}}
```

- That's `pi`'s bash tool, abbreviated
- No registration, no capability grant: it's **prompt**
- Remove it from the request; the model has never heard of bash

The Transcript, With Tools

Module 1′s picture, with two new kinds of row:

system	“You are…” — and the tool schemas live here too
user	“What’s in src/?”
assistant	tool_call ⟶ `bash("ls src/")`
tool	`auth.py  routes.py  util.py`
assistant	“Three files. auth.py handles…”

The ledger only ever appends: a tool call is two more rows (request, then result), never an edit. (Lab 2.5 hands you the eraser anyway.)

The Harness

The harness is everything around the model — the program you actually run:

- **The loop**: every arrow in the loop diagram is harness code
- **Tools**: described to the model in the request; **executed** by the harness
- **Context assembly**: the system prompt, plus what stays in the window
- **Permissions**: what the model may do without asking you

The model supplies judgment; the harness supplies everything else.

Model × Harness

An agent is a pairing, and the harness is the half you control:

- Same harness, different model: you change one string (Lab 1.1)
- Same model, different harness: a different agent — the harness decides what the model even sees
- The model is rented and opaque; a harness can be owned, read, and patched
- Every safety property lives here, because the model can only **request**

Model quality is half the product. The other half is this program — in Lab 2.5 you modify it.

The Harness Landscape

Same loop, different philosophies:

- **Cursor** — IDE-embedded, autocomplete-first; agent features bolted onto an editor
- **Claude Code** — polished and capable, but vendor-tied: one lab's models, a closed harness
- **`pi`** — small, open, hackable, model-agnostic — and you can read all of it

Why this class uses `pi`: in Lab 2.5 you extend the harness itself. The others don't hand you the loop.

`pi`

Our harness for this class.

- Terminal-first: run it in a directory, give it a task
- Built-in tools: read, write, edit, bash, and more
- Extensible: skills, custom tools, subagents (second half of this module)
- Points at the class proxy: same API you used in Lab 1.1
- Capability control: tool allowlists and environment boundaries; you'll compare restricted and autonomous runs in Lab 2.3

Watch the loop as it runs: every tool call and result is visible. That transparency is a feature; use it.

Working With Agents

Supervising

An agent is a fast junior colleague with no long-term memory and boundless confidence.

- **Watch the loop**: tool calls tell you what it thinks it's doing — it's the loop diagram, running live in your terminal
- **Interrupt early**: looping, drifting, or brute-forcing means steer now, not after ten more calls
- **Verify**: it says "done"; the tests say the truth

The failure mode isn't usually the agent doing something crazy. It's the agent doing something **plausible and wrong**, confidently.

Steering

- Small, verifiable steps beat grand missions
- State constraints up front (files, tools, scope) — not after violation
- When it flails: don't argue in the same session. Restate the task with what you learned, in a **fresh** session.

Context rot from Module 1 applies double here: an agent generates its own context at machine speed, including its own mistakes.

Where Agents Pay Off

Strong:

- Multi-step tasks with a verifiable endpoint (tests, checkers)
- Exploring unfamiliar code, repetitive refactors, glue work
- Anything where iteration against feedback beats first-try correctness

Weak:

- One-shot judgment with no feedback signal
- Tasks you can't verify — you just inherit confident wrongness
- Ten-second tasks: loop overhead exceeds the work

Extending Agents

Why Extend?

An agent out of the box knows nothing about **your** world.

Four mechanisms, in order of increasing machinery:

1. **Skills** — packaged knowledge
2. **Custom tools** — new capabilities
3. **MCP servers** — external tool providers
4. **Subagents** — delegation and parallelism

The most common mistake: heavy machinery where a paragraph would do.

Skills

A skill is packaged knowledge: instructions + resources the agent loads when relevant.

- “Here’s how to interpret this tool’s output”
- “Here’s our team’s bug report format, with a template”
- “Here’s the workflow for X, step by step”

Use a skill when the agent **could** do the task if someone explained it once. You are writing that explanation down — documentation the agent actually reads.

Custom Tools

A tool is a new capability: code the harness runs on the model's request.

- Wrap an internal CLI or API
- Give structured access to something bash-able only painfully

Use a tool when the agent **cannot** do the thing, or does it too unreliably to trust.

Skill vs. tool in one line: **skill = knowledge, tool = capability.** Lab 2.4 has you build both skill flavors; the second ships examples and its own CVSS scoring script — knowledge can carry code.

MCP Servers

Model Context Protocol: a standard for tool servers that any MCP-aware harness can connect to.

- A separate process exposing tools; the harness bridges them to the model
- Good: shared tooling across teams and harnesses, vendor integrations
- Cost: another running service, another trust decision

Rule of thumb: if a CLI wrapper does the job, skip MCP — and vet third-party servers like dependencies (Module 4).

In Lab 2.2 you build one: your calculator tools, lifted into a server any MCP client can discover. Stock `pi` deliberately has none — both sides of the trade, first-hand.

MCP Under the Hood

Not magic either: a client-server protocol for tool **discovery**.

- **Wire**: JSON-RPC 2.0 over stdio (or HTTP)
- **Client**: the harness. **Server**: the process that owns the tools.
- Lifecycle: `initialize` handshake → `tools/list` (server advertises name + description + schema) → `tools/call` (request/response, repeat)

Name, description, JSON schema: exactly what you already inject into the request by hand. MCP standardizes the injection.

`tools/call`, On the Wire

```
{"jsonrpc": "2.0", "id": 7, "method": "tools/call",
 "params": {"name": "add", "arguments": {"a": 2, "b": 2}}}
```

```
{"jsonrpc": "2.0", "id": 7,
 "result": {"content": [{"type": "text", "text": "bread"}]}}
```

(Your Lab 2.1 calculator, lie intact; protocols don't launder tool output.)

- The harness turns `tools/list` responses into the same `tools` parameter as always; the model can't tell an MCP tool from a local one
- MCP is plumbing for **discovery**, not new model capability

Subagents

A subagent is **just another tool**: it spawns a child agent with a clean context, and the parent reads its output like any tool result.

On the loop diagram: a second loop hiding inside the **execute** box.

- **Context isolation**: exploration mess stays out of the parent
- **Parallelism**: fan out independent subtasks
- **Specialization**: different instructions per subtask

When: the subtask is self-contained and its journey doesn't matter, only its answer. Don't delegate what needs the parent's full context.

Bash: The Universal Tool

Models only speak text. The shell is a text-in, text-out interface to nearly everything a computer can do: built for humans typing, perfect for models emitting.

- One `bash` tool subsumes most bespoke tools and MCP servers you'd write
- `grep`, `curl`, `git`, `nmap`: already tools, already in the training data

The caveat: autonomy with bash is safe **here** because your instance is disposable and sandboxed. At home, that's the permission-modes discussion: match autonomy to blast radius.

The Decision Framework

Situation	Reach for
agent could do it if told how, once	prompt it
…and you'll need it again	skill
agent can't do it / needs guaranteed path	tool
same tools, many harnesses or teams	MCP server
self-contained subtask, parallel or noisy	subagent

Start at the top; move down only when the cheaper option fails. Worth repeating: most bespoke tools are just bash, and if a CLI wrapper does the job, **skip MCP**.

Using `pi` to Extend `pi`

The agent can write its own extensions.

- "Wrap this CLI as a tool" is a well-defined coding task — exactly what agents are good at
- Have it run the tool, hit the rough edges, and fix them: the feedback loop again

You'll do this in Lab 2.5, and it's the habit to leave with: **every repeated explanation becomes a skill; every awkward workaround becomes a tool.**

Wrap-Up

What You Should Take Away

- Agent = LLM + tools + loop; the harness holds all the actual power
- The model only requests; execution and permissions live in the harness
- Supervise the loop; interrupt early; verify claims of “done”
- Skill = knowledge, tool = capability; prompt first, machinery later
- The agent can build its own extensions — make it do the work

Labs 2.1–2.5: hand-build an agent, serve its tools over MCP, drive `pi`, teach it skills — then rewrite its memory