Agents: Using and Extending Them

Agentic Security — Module 2

What Is an Agent?

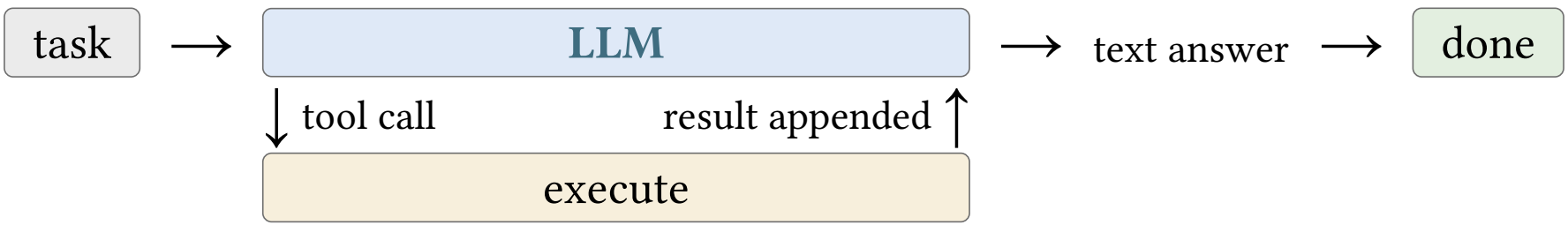
The Definition

An **agent** is an LLM given tools and the ability to run in a loop. A **harness** is a program that gives an LLM these things.

1. The model receives a task and the tools it can use
2. The model responds with either a tool call or an answer
3. If it's a tool call, the harness evaluates it, adds the result to the context, and the model responds again (go to 2)
4. If it's an answer, the harness stops and prints it

That's it. Everything else — "autonomy", "reasoning", "planning" — emerges from the combination of a good model and a harness.

The Loop Illustrated



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

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 turn	errors get fed back and fixed

The feedback cycle of the loop is what lets an agent recover and learn. It can run tests, read failures, fix them, and repeat until success.

Tool Calling

How It Actually Works

A model calls a tool by emitting specific tokens a provider recognizes. For example:

```
<call:bash>{"command":"ls"}</call>
```

The provider can use a regular expression to parse this:

```
<call:(?P<name>[a-zA-Z0-9_-]+)>(?P<args>\{.*?\})</call>
```

- `<call:` and `</call>` mark the tool call boundaries
- `(?P<name>...)` captures the tool name: `bash`
- `(?P<args>...)` captures the arguments: `ls`

How It Actually Works

```
{"role": "assistant", "tool_calls": [{"id": "call_a1b2c3",
 "function": {"name": "bash",
              "arguments": "{\"command\": \"ls\"}"}}]}
```

- The provider parses the model's emitted tokens into this shape and sends it to the client

How Does the Model Know About Tools?

A list of tools is sent by your harness to the model with every request.

For example, here is a `bash` tool definition:

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

The Transcript

system	“You are Claude. […] You can call these tools: […]”
user	“What’s in src/?”
assistant	tool_call ⟶ `bash("ls src/")`
tool	`auth.py  routes.py  util.py`
assistant	“Three files. auth.py handles…”

As always, the transcript is **append-only**. The harness is in charge of managing the messages array.

⏸ PAUSE — Lab 2.1

Evolve your chatbot into an agent by giving it custom tools.

Model Context Protocol (MCP)

What is MCP?

Model Context Protocol: a standard for exposing tools.

- An MCP server is a separate process exposing tools. It can live on your computer or on the Internet.
- An MCP-aware harness knows how to connect these tools to models
- Pros: shared tooling across teams and harnesses, vendor integrations
- Cons: another running service, another trust decision

Rules of thumb: if a command-line tool does the job, skip MCP. Vet third-party MCP servers carefully for security.

MCP Under the Hood

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)

MCP offloads tool management to a server, which is often convenient but means **you must trust the server**.

The Harness Landscape

- **Cursor** — graphical app. Comes with advanced manual text editing features.
- **Claude Code** — polished and capable, but tied to Anthropic models.
- **`pi`** — minimal, open-source, hackable, model-agnostic.

`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, extensions
- Automatically configured for the class, using the same API you used in Labs 1.2 and 1.3

⏸ PAUSE — Lab 2.2

Take `pi` for a test drive.

Working With Agents

Supervising

An agent is like a fast but naive junior colleague with no long-term memory and way too much confidence.

- **Interrupt early**: clear up confusion and misunderstandings early if you can
- **Verify**: it will say "done" whether or not it's actually done. Use tests.

Today, agents doing something obviously crazy is relatively rare.

The actual danger is confident mistakes that look correct at first.

Steering

- Small, verifiable steps are better than huge missions
- State your constraints up front
- Never argue with the model. Instead, start a fresh session and try again.

Where Agents Pay Off

Strong:

- Multi-step tasks with a verifiable endpoint (tests, checkers)
- Exploring and explaining unfamiliar code
- Doing repetitive work

Weak:

- One-shot judgment with no feedback signal
- Tasks you can't verify — you have no way to catch it being wrong
- Tasks that require significant background knowledge

Extending The Harness

Why Extend?

A harness's built-in tools often aren't enough to do the job well.

Harnesses commonly have several mechanisms to extend the regular tool-calling paradigm:

1. **Skills** — packaged knowledge
2. **Custom tools/MCP servers** — new capabilities
3. **Subagents** — delegation and parallelism

Skills

A skill is a dynamically loaded bundle of knowledge.

Harnesses give agents a list of available skills. Agents can read more about a skill if they choose to.

Useful for:

1. Explaining specific procedures
2. Encoding knowledge
3. Bundling custom scripts with instructions on how to use them (you'll do this in Lab 2.3)

⏸ PAUSE — Lab 2.3

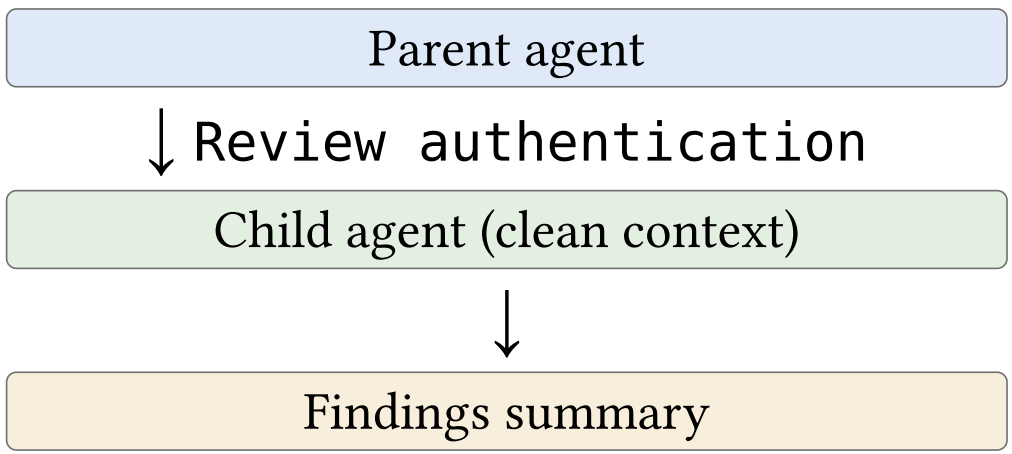
Teach `pi` a new skill.

Subagents

A subagent is a separate agent the parent calls like a tool.

Example: review authentication code.

- Starts with a clean context
- Uses its own tools
- Returns a summary



Subagents

Benefits:

- **Context isolation**: exploration mess stays out of the parent's limited context
- **Parallelism**: fan out independent subtasks to get more work done faster
- **Specialization**: different instructions per subtask

Bash: The Universal Tool

LLMs are text-in, text-out. So is Bash, the most common Linux shell.

Bash was written by and for humans, but it turns out LLMs are also pretty good at it.

- One Bash tool can replace many custom tools and MCP servers
- Allows access to useful commands like `grep`, `curl`, `ls`, `git`

Bash's strength is also its danger. Giving an agent access to Bash is risky unless sandboxed properly.

The Decision Framework

Situation	Reach for
agent could do it if told, once	prompt it
…and you'll need it again	skill
agent can't do it on its own	tool
you want to share or use others' tools	MCP server
your task has lots of subtasks	subagent

⏸ PAUSE — Lab 2.4

Use `pi` to extend `pi`.

Wrap-Up

Takeaways

- Agent = LLM + tools + loop.
- LLMs don’t act, they ask. Harnesses do the work.
- Verify everything.
- Agent workflows are very customizable.

Next module: Agentic Coding