WORKSHOP OVERVIEW

Building AI Agents with Claude Code

From "what is an agent?" to shipping autonomous agents on your own machine. Builds on Basics + Skills.

Table of contents

MODULE 1

What is an agent?

Concept · 30 min · No keyboard yet — read, watch, sketch

1.1

The progression: prompt → skill → agent

⏱ 5 min

You already know the first two. The third is the leap.

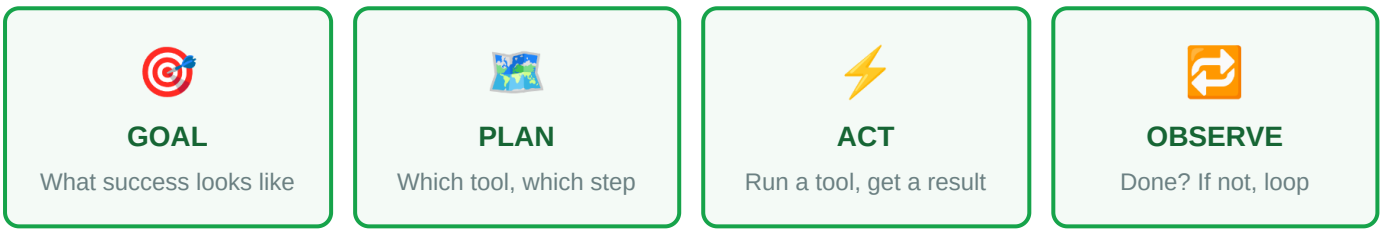
Type	What it does	Who decides next step?
Prompt	One question, one answer	You — every turn

Type	What it does	Who decides next step?
Skill	Reusable expert behavior on one task	You — invoke when needed
Agent	Pursues a goal across many steps	**The agent itself**, within your boundaries

1.2 The agent loop

⏱ 10 min · learn this diagram

Every agent — no matter how complex — runs this loop:



The three ingredients

- **Tools** — what the agent can *actually do* in the world (read files, search web, call APIs)
- **Memory** — what it remembers across the loop (context, intermediate results)
- **Autonomy** — how many steps it can run before checking in with you

1.3 When NOT to build an agent

⏱ 5 min

⚠️ Reach for a prompt or skill first

Most jobs don't need a loop. Use an agent only when:

- The number of steps depends on what's discovered along the way
- The work runs unattended (scheduled, triggered by an event)
- Multiple tools must be combined dynamically

Otherwise: a skill is cheaper, faster, and easier to debug.

1.4 Live demo: same task, three ways

⏱ 10 min · instructor demo

Watch the same task — "summarize my recent meeting notes" — done as:

1. A **prompt** in Claude.ai (paste notes, get summary)
2. A **skill** (`meeting-notes.md`) — Claude knows your format every time
3. An **agent** — reads your `~/Notes` folder, finds yesterday's, summarizes, writes back

Notice: each one solves a different shape of problem.

MODULE 2

First agent: File Organizer

Hands-on · 45 min · Build a real subagent that organizes your Downloads folder

2.1 Open a workshop project

⏱ 3 min

In Claude Code, open the folder `~/agents-workshop` (create it if needed). All work today happens here.

2.2 Create the subagent file

⏱ 5 min

Subagents in Claude Code live in `.claude/agents/`. Ask Claude to scaffold one for you:

PROMPT 1 — TYPE THIS:

"Create a subagent at `.claude/agents/file-organizer.md`. Its job is to scan a folder I point it to, group files by type (images, docs, archives, code, video, other), and move them into typed subfolders. Allow only Read, Glob, Bash. Default to dry-run mode unless I pass `execute=true`."

Claude will create the file. Open it. You'll see a YAML frontmatter and a system prompt body. Read it — you should be able to follow every line.

What's in a subagent file

- **name & description** — how it shows up in your agent list
- **tools** — the allow-list. Anything not listed, the agent can't use
- **system prompt** — the agent's persona, rules, and how it handles edge cases

2.3 Run it dry-first

⏱ 10 min

PROMPT 2:

"Run the file-organizer subagent against `~/Downloads` in dry-run mode. Show me the plan but don't move anything yet."

You'll see Claude:

1. Glob the folder
2. Group files by extension/type
3. Print the proposed move plan
4. Stop, and ask if you want to proceed

> ⚠️ **Read the plan before approving**
>
> This is the moment that separates "I trust this agent" from "I just lost my files." Always read the plan. Always.

2.4 Run it for real

⏱ 7 min

PROMPT 3:

"Looks good. Run it with `execute=true`."

Watch the trace. Claude will move files one by one, reporting each move. When it's done, you'll have a clean Downloads folder.

2.5 Debug a misfire (on purpose)

⏱ 12 min

Now — deliberately — break it. Add a file with no extension to `~/Downloads` and run the agent again. It will misfire. Read the trace. Find the gap in the system prompt.

PROMPT 4 — FIX THE AGENT:

"The agent doesn't handle files without extensions. Update `file-organizer.md` so unknown files go into an `unsorted/` subfolder. Re-run dry-run to verify."

✅ **Module 2 complete when:**

- [ ] You have `.claude/agents/file-organizer.md` on disk
- [ ] You ran it dry, then live, on real Downloads
- [ ] You debugged at least one misfire
- [ ] You can explain what each line of the agent file does

MODULE 3

Tools beyond the filesystem

Hands-on · 45 min · Connect agents to the web, MCP servers, and custom tools

3.1 Built-in tools you already have

⏱ 5 min

Tool	Use it for	Watch out for
WebFetch	Reading a known URL	Rate limits, bot blockers
WebSearch	Finding URLs you don't know yet	Always verify with WebFetch
Bash	Anything the shell can do	Highest blast radius — hook it
Read / Write / Edit	Files in your project	Path scope (use hooks)

3.2 MCP servers — connect to real systems

⏱ 15 min

MCP (Model Context Protocol) lets your agent talk to GitHub, Slack, Calendar, Linear, Notion, Gmail, and dozens more. Install one now:

```
# Example: install the GitHub MCP server
claude mcp add github -- npx -y @modelcontextprotocol/server-github
# You'll be prompted for a personal access token
```

Once installed, restart Claude Code. New tools (`github_list_prs` , `github_get_pr` , etc.) appear automatically.

3.3 Build a GitHub triage agent

⏱ 15 min

PROMPT:

"Create `.claude/agents/pr-triage.md`. It uses the GitHub MCP tools to list open PRs in a repo I name, fetches the title, author, and last update for each, then writes a summary table sorted by staleness. Read-only — no comments, no merges."

Test it on a real repo you maintain.

3.4 Custom tools via the Agent SDK

⏱ 10 min · concept + 5-line example

When no MCP server exists for what you need, write a custom tool. Example: a tool that hits your company's internal API.

```python
# Pseudocode — Python Agent SDK
from claude_agent_sdk import tool, agent

@tool("check_inventory")
def check_inventory(sku: str) -> dict:
    """Returns stock for a given SKU from internal API."""
    return requests.get(f"https://api.acme.internal/sku/{sku}").json()

agent(tools=[check_inventory], goal="Reorder anything below 10 units")
```

Tool selection mental model

When picking a tool, ask:

- **I/O cost** — how slow / expensive is one call?
- **Blast radius** — what's the worst thing it can do?
- **Confirmation** — should the agent stop and ask before using it?

MODULE 4

Multi-step agent: Research Brief

Hands-on · 60 min · Build an agent that decomposes its own work

4.1 The plan-then-execute pattern

⏱ 8 min

The simplest pattern that handles real complexity: **first plan, then execute, then self-review.**

1. **Plan** — agent writes the steps it will take, before any tool calls
2. **Execute** — runs the plan, one step at a time
3. **Self-review** — checks the output against the goal, revises if needed

4.2 Build the agent

⏱ 10 min

PROMPT 1 — SCAFFOLD:

"Create `.claude/agents/research-brief.md`. Given a topic, produce a 1-page brief with: 5 sources (URLs, titles, dates), claims with citations `[1]` ... `[5]`, and a one-paragraph synthesis. Tools: WebSearch, WebFetch, Write. Plan-then-execute pattern. Self-review for citation accuracy before returning."

4.3 Run on a real topic

⏱ 20 min

PROMPT 2:

"Use research-brief on the topic 'agentic coding 2025'. Save the output to `briefs/agentic-coding-2025.md`."

Watch carefully. The agent should:

1. Print its plan first (5 search queries, what it expects to find)
2. Execute each search
3. Read the top hits
4. Self-review: "did I cite every claim?"
5. Revise if needed, then write the file

4.4 Force better citations

⏱ 10 min

If your brief has uncited claims, fix the agent — not the output. Add to the system prompt:

EDIT THE AGENT:

"Update research-brief.md: any claim without a citation marker `[N]` must be removed during self-review. No exceptions."

4.5 When the agent should ask vs. proceed

⏱ 12 min

Decision rules to bake in

- **Ambiguous goal** → ask once at the start, then proceed
- **Conflicting sources** → present both with citations, don't pick
- **Cost spike** (more than N searches) → stop and report
- **No results** → return a "couldn't find" brief, don't fabricate

✅ Module 4 complete when:

- ☐ You have a saved 1-page brief in your `briefs/` folder
- ☐ Every claim has a citation marker
- ☐ You ran it on at least 2 different topics
- ☐ You can explain plan-then-execute without looking

MODULE 5

Production concerns

Hands-on · 45 min · Agents you'd trust to run while you're not watching

5.1 Budgets — caps on cost and runaway loops

⏱ 5 min

Budget	What it caps	Default to set
Max steps	Loop iterations	20–50
Max tool calls	External calls	30
Time cap	Wall-clock	5–10 min
Token cap	Context spend	Project-level

Add to any agent's system prompt: *"Stop and report if you exceed 30 tool calls or 8 minutes of work."*

5.2 Hooks as guardrails

⏱ 15 min

Hooks run shell commands before/after tool calls. Use them to *enforce* rules the agent could otherwise bend.

ADD A PRE-WRITE HOOK:

"Add a `PreToolUse` *hook to my project's* `.claude/settings.json` *that blocks any Write or Edit outside* `~/agents-workshop/`*. Hook fails the call if the path doesn't match."*

```
// .claude/settings.json — example shape
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Write|Edit",
```

```
        "hooks": [{
          "type": "command",
          "command": "check-path-allowed.sh"
        }]
      }
    ]
  }
}
```

5.3 Idempotency & retries

⏱ 10 min

Make every tool call safe to repeat

- **Check before write** — does the file already exist? skip or update
- **Use unique IDs** — task-id-{date} so repeated runs don't double-create
- **Mark progress** — write a `.done` file after each completed step
- **Resume, don't restart** — if the agent dies mid-run, the next run picks up where it stopped

5.4 Observability — read the trace

⏱ 8 min

Every agent run leaves a trace. Find it:

```
# macOS / Linux
ls ~/.claude/projects/<project>/

# Open the latest run
cat ~/.claude/projects/<project>/<latest>.jsonl | jq
```

When an agent misfires, share that file. It contains every prompt, every tool call, every output. It's your debugger.

5.5 When humans must stay in the loop

⏱ 7 min

⚠️ Always require human approval for…

- Sending external messages (email, Slack, customer-facing)

- Writing to shared infrastructure (production DB, CI config, prod branches)
- Spending money (calling paid APIs, charging cards, deploying)
- Anything destructive (delete, force-push, drop)

Implement via hooks — don't trust the agent's "I'll be careful" promise alone.

MODULE 6

Capstone — pick & ship

Hands-on · 60 min · Choose a template, customize for your real life, demo it

Pick **one** template. Customize it with your real data. Run it end-to-end. Demo in 90 seconds.

6.1 Template A — Inbox Triage

Goal · drafts replies in your tone

SCAFFOLD:

"Create `.claude/agents/inbox-triage.md`. Uses the Gmail MCP tools to read the last 20 unread messages, classify each (reply / archive / forward / wait), and for reply candidates draft a response in my tone. Outputs a markdown table — never sends. I send manually."

6.2 Template B — Daily Digest

Goal · morning brief on schedule

SCAFFOLD:

"Create `.claude/agents/daily-digest.md`. At 7am, fetches the latest from 3 sources I list (RSS feeds, hacker news, a Slack channel), writes a 200-word brief, saves to `~/digest/YYYY-MM-DD.md`. Read-only. Skip the day if nothing new."

6.3 Template C — PR Reviewer

Goal · checklist-based review comments

SCAFFOLD:

"Create `.claude/agents/pr-reviewer.md`. For a given PR URL, fetches the diff, runs through our team checklist (edge cases, tests, naming, performance), and produces a comment-ready review in markdown. Posts to GitHub only if I confirm with `post=true`."

6.4 Template D — Calendar Coordinator

Goal · time-block your week

SCAFFOLD:

"Create `.claude/agents/cal-coordinator.md`. Reads my calendar for the next 7 days, finds open 90-minute blocks, drafts proposed deep-work sessions matching priorities I list. Flags any conflicts with existing events. Outputs a plan — does not modify the calendar."

6.5 Template E — Research Companion

Goal · scaled-up Module 4

SCAFFOLD:

"Adapt the research-brief agent into `.claude/agents/research-companion.md`. Maintains a topics list. Daily, picks the topic with the oldest brief and refreshes it. Saves history so I see what changed week over week."

6.6 Demo (90 seconds each)

⏱ shared show & tell

Your demo answers:

- [ ] What's the goal of your agent?
- [ ] Show the trace of one run
- [ ] What's one thing it got wrong, and how did you fix it?
- [ ] What would you build next on top of it?

CHEAT SHEET

Take this home

Reference · keep this on your desk

The agent loop

GOAL → PLAN → ACT → OBSERVE → loop or stop. If your agent isn't doing all four, it's a skill in disguise — make it one.

Subagent file template

```
---
name: my-agent
description: One sentence — when to invoke this agent
tools: Read, Write, Bash, WebSearch
---

You are an agent that <goal>.

PLAN before acting. State your plan in one paragraph.
ACT one step at a time. After each tool call, ask yourself: am I done?
STOP if you exceed 30 tool calls, 8 minutes, or hit anything ambiguous.

NEVER:
- Write outside <allowed paths>
- Send external messages without explicit approval
- Continue past a destructive action without confirmation
```

Tool selection chart

Need	Tool	Approval needed?
Read project files	Read, Glob	No
Modify project files	Write, Edit	Hook-enforced scope
Run shell commands	Bash	Yes — high blast radius
Read the web	WebFetch, WebSearch	No
Talk to GitHub / Slack / Cal	MCP server	Per-action — read free, write gated

Need	Tool	Approval needed?
Custom internal API	Agent SDK custom tool	Define per call

Common failure modes & fixes

Symptom	Likely cause	Fix
Agent loops forever	No stopping condition	Add max-steps, explicit "done" criteria
Agent fabricates results	Tool failed silently	Force the agent to print tool output before claiming success
Agent over-edits	Scope too broad	Tighten path allow-list, add PreToolUse hook
Slow / expensive runs	Too many web calls	Cap searches, cache fetched URLs in a temp file
Inconsistent output format	Output spec implicit	Add a literal example of the expected output to the system prompt

10 agent ideas to build at home

1. **Receipt sorter** — reads PDFs in ~/Receipts, extracts amounts, builds a monthly CSV
2. **Slack standup writer** — reads your last 24h of commits + PRs, drafts a standup post
3. **Bookmark librarian** — periodically de-dupes your browser bookmarks, tags by topic
4. **Deal alert** — watches a list of products, pings you when prices drop below targets
5. **Code-archaeology** — given a function, traces its callers and writes a one-pager on its history
6. **Meeting prep** — for tomorrow's calendar, drafts 3-bullet prep notes per meeting
7. **Inbox archivist** — auto-archives newsletters, labels promos, surfaces only personal mail
8. **Doc gardener** — scans your repo for stale README/docs, opens PRs with refresh suggestions
9. **Newsletter drafter** — turns a week of your tweets/posts into a newsletter draft
10. **Onboarding buddy** — for a new repo you cloned, writes a 1-page "where things live" guide