Modern AI Systems: Agents and System Optimizations

Agent From a User's Perspective II: Multi-agent and Best Practice

Juncheng Yang

Harvard John A. Paulson School of Engineering and Applied Sciences

HARVARD UNIVERSITY
MaDSys
Measurement and Design of Systems Lab

Update and Reminder

- Assignment 1 due
- Assignment 1 sharing signup
- Add me and TFs to your repo

New TF: Charles Wang

Recap

- Agent loops
- State management
  - context as working memory: each user prompt, LLM response, tool call result is appended
  - agent memory: persistent across sessions
    - CLAUDE.md (AGENTS.md)
    - auto memory
- Tools
  - built-in tools
  - extending Claude Code capability: skill, MCP, hook, plugin

Plan for Today

- Case studies: Superpowers and Caveman
- Become a 100x researcher and engineer with multiple agents
- Programming / Software 3.0: how programming has evolved
- How people use OpenAI Codex
  - two reports from June 2026 and September 2026

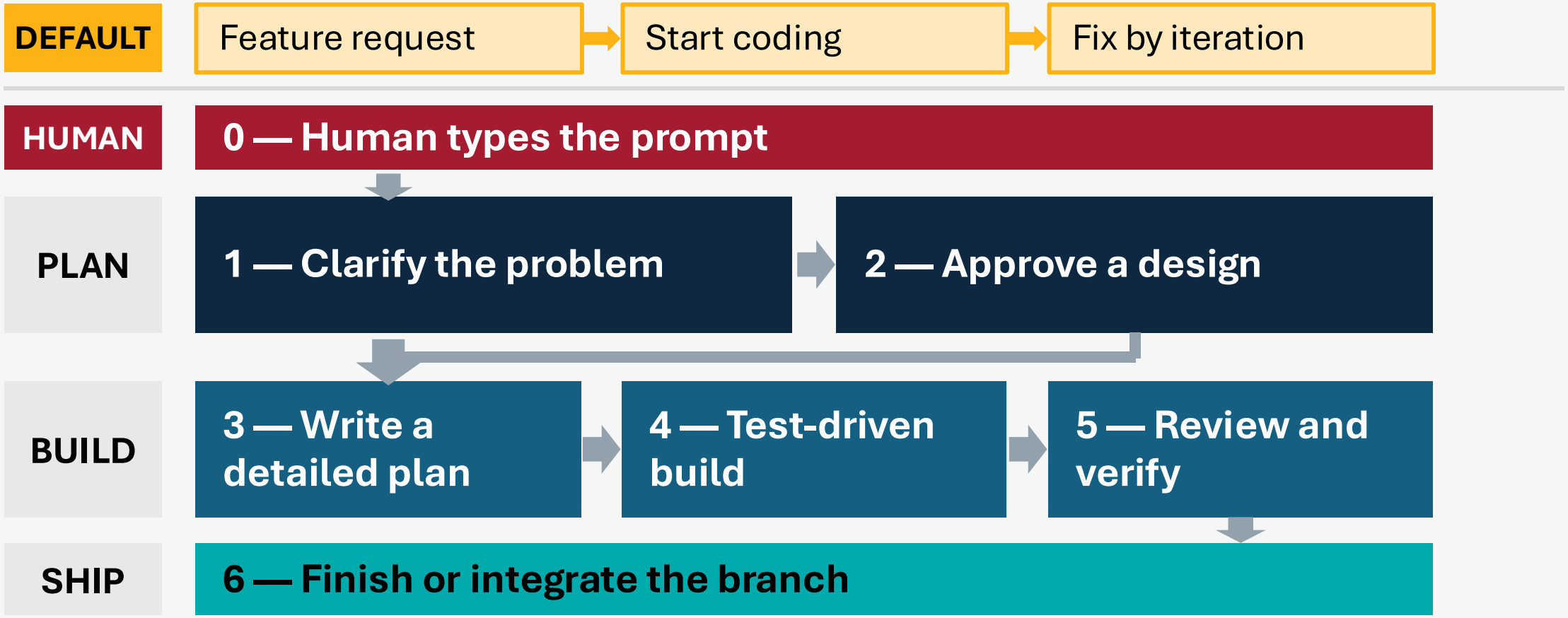
```
/plugin install superpowers@claude-plugins-official
```

Case Study: Superpowers

A software-development plugin

https://github.com/obra/superpowers

Same Model, Different Workflow



Both agents share the same model and the same tools — only the instructions differ

OPTIONAL — extra reading, not required

Nine Skills in Superpowers

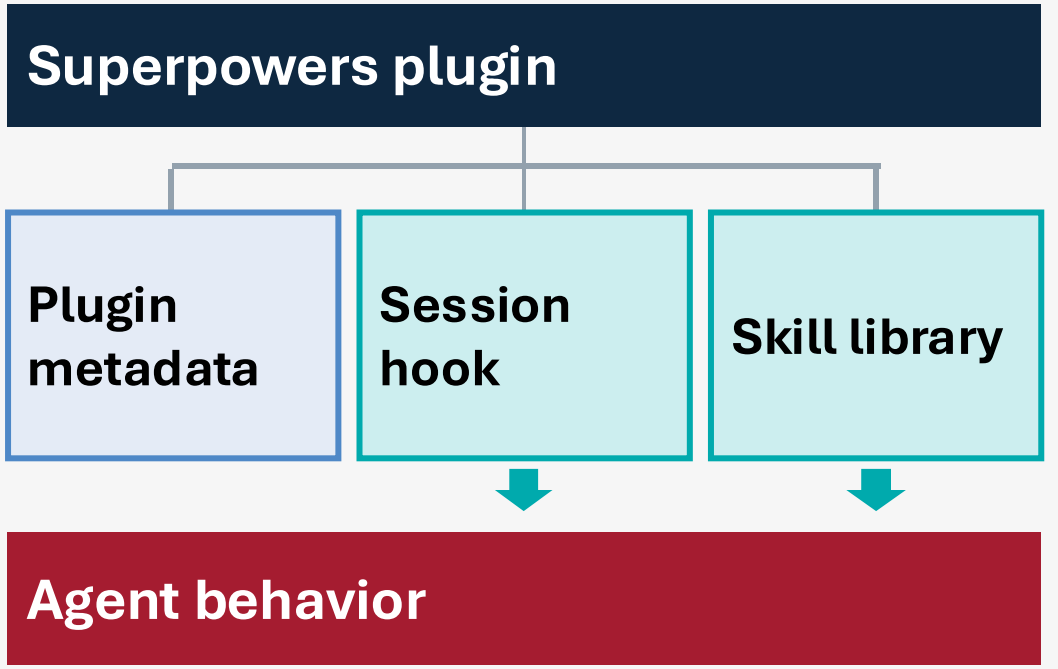
Skill	Responsibility
`using-superpowers`	Rules for discovering and invoking skills
`brainstorming`	Turns a request into an approved design
`writing-plans`	Decomposes the design into concrete tasks
`using-git-worktrees`	Creates an isolated workspace
`test-driven-development`	Enforces red, green, refactor
`systematic-debugging`	Finds root causes before proposing fixes
`requesting-code-review`	Reviews implementation against the plan
`verification-before-completion`	Requires evidence before reporting success
`finishing-a-development-branch`	Merge, pull request, retention, or cleanup

Superpowers Architecture

Repository layout

```
superpowers/
├── .claude-plugin/   plugin.json
├── hooks/            hooks.json
│                   session-start
├── skills/
│     using-superpowers/
│     brainstorming/
│     writing-plans/
│     test-driven-development/
└── tests/
```

Component map



A plugin is the distribution container — skills and hooks supply the behavior

Anatomy of a Skill

Directory

```
skills/
└── brainstorming/
      SKILL.md
      visual-companion.md
      scripts
```

Front matter is metadata. The body is the procedure the agent follows.

SKILL.md (simplified)

```
---
name: brainstorming
description: Use before creative work.
---
## Workflow
1. Inspect the project
2. Classify the scope
3. Clarify requirements
4. Compare approaches
5. Present a design
6. Wait for user approval
7. Invoke writing-plans
```

What a SKILL.md Contains

Six components of a skill file

Component	Function
`name`	Stable identifier
`description`	When the skill applies
Instructions	The procedure to follow
Checkpoints	Approval before proceeding
References	Detail only when needed
Scripts	Deterministic operations

What a process skill can define

- **Entry conditions**
- **Required actions**
- **Prohibited actions**
- **Human approval gates**
- **Verification requirements**
- **The next applicable skill**

Each skill guides agent behavior and optionally names its successor

Why Skills Alone Are Not Enough

Installing skills makes them available — availability alone does not guarantee the agent check them

hooks/hooks.json

```
{
 "hooks": {
  "SessionStart": [{
   "matcher": "startup|clear|compact",
   "hooks": [{ "type": "command",
    "command": "run-hook.cmd",
    "async": false }]
  }] } }
```

What the hook guarantees

- **Deterministic timing, every session**
- **Runs before the agent acts**
- **Re-runs after clear and compaction**

OPTIONAL — extra reading, not required

The Session-Start Script

hooks/session-start (simplified)

```
#!/usr/bin/env bash
set -euo pipefail

ROOT="$(plugin_root)"
skill="$ROOT/skills/using-
       superpowers/SKILL.md"
content="$(cat "$skill")"

printf '{"hookSpecificOutput": {
 "hookEventName": "SessionStart",
 "additionalContext": "%s" }}' \
 "$(escape_json "$content")"
```

Execution steps

1 — Locate the installed plugin

2 — Read the bootstrap SKILL.md

3 — Escape the Markdown for JSON

4 — Return it as additionalContext

5 — Claude Code adds it to context

End-to-End Walkthrough

Runtime sequence: “Add authentication”

- Claude Code → starts the session
- Hook → reads the bootstrap skill
- Agent → receives the selection policy
- User → “Add authentication”
- Agent → loads brainstorming, clarifies
- User → approves the design
- Agent → loads writing-plans

Later stages

- Implementation plan
- Isolated worktree
- Test-driven build
- Code review
- Verification
- Branch integration

OPTIONAL — extra reading, not required

Design Lessons From Superpowers

Design question	Superpowers' answer
How should complex behavior be represented?	**Small, composable skills**
How does the agent remember to use them?	**A bootstrap skill via a session hook**
How is context kept manageable?	**Load task skills only when relevant**
How do stages coordinate?	**Each skill names its gates and successor**
How do users install the system?	**Package everything as a plugin**
Does every plugin need commands or MCP?	**No — add them only when needed**

Caveman case study

Five extension mechanisms — plus a local proxy

```
claude plugin marketplace add JuliusBrussee/caveman
claude plugin install caveman@caveman
```

Why compress an agent's tokens?

Context fills up

Logs, diffs and tool output crowd out the work.

Every turn costs

Tokens are paid for and waited on, both ways.

Compaction loses detail

Trimming context late drops what mattered.

BEFORE — 786 lines of test output

```
$ npm test
FAIL  src/api/client.test.ts
  ● client › retries on 5xx
    connect ETIMEDOUT 10.0.3.8:443
      at client.ts:88
… 776 more lines …  3 failed, 214 passed
```

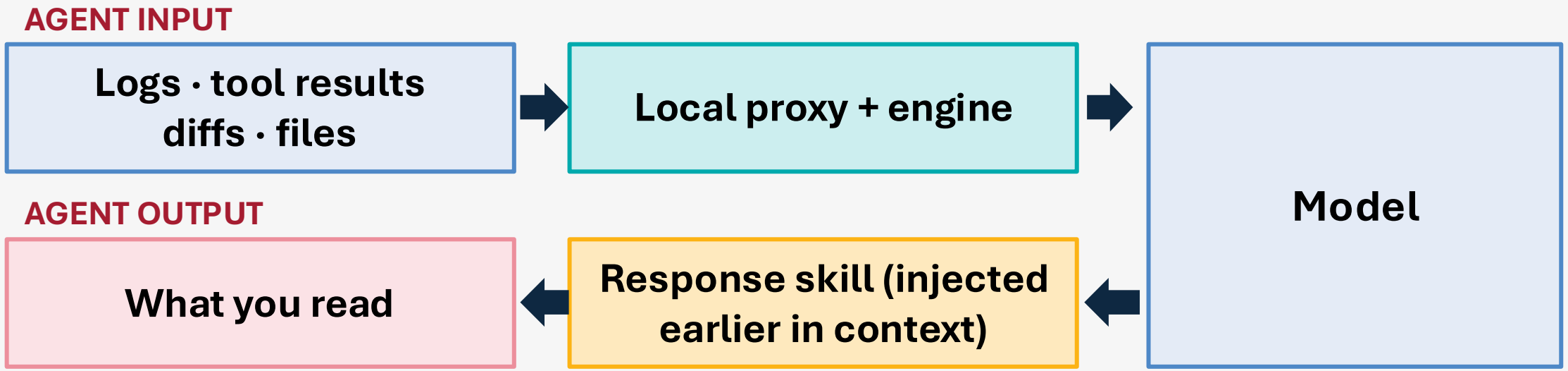
AFTER — what the model reads

```
3 tests fail: ETIMEDOUT at client.ts:88
```

Compress both directions — and keep the original available for retrieval.

Caveman compresses tokens in two directions

`/caveman ultra` then: “Run the tests and explain the failure.”



The skill changes how the model writes. The proxy changes what it reads.

Reported: 65% fewer output tokens
The skill itself adds input tokens, so whole-session savings are lower.

OPTIONAL — extra reading, not required

Walk through an example

`/caveman ultra` then: “Run the tests and explain the failure.”

	WHAT HAPPENS, IN ORDER
1	Hook stores ultra mode
2	Hook injects the concise-response reminder
3	Agent executes the command (from LLM)
4	Caveman (proxy) compresses the noisy test output
5	Model receives reduced context
6	Model returns the concise diagnosis
7	MCP retrieves the original on demand

Plugin packaging

REPOSITORY STRUCTURE

```
caveman/
├─ .claude-plugin/plugin.json
├─ skills/
│   caveman, -commit, -review,
│   -compress, -stats
├─ commands/
├─ src/hooks/
├─ mcp/
├─ proxy/
├─ engine/
└─ agents/
```

PLUGIN MANIFEST

```
{
  "name": "caveman",
  "hooks": {
    "SessionStart":
      ["caveman-activate.js"],
    "UserPromptSubmit":
      ["caveman-mode-tracker.js"]
  }
}
```

The response skill makes output less verbose

SKILLS/CAVEMAN/SKILL.md

```
---
name: caveman
description: Compressed mode
---
# Caveman response mode
Preserve: technical terms, commands, paths, code, errors,
numbers, units, negation
Remove: filler, repetition, pleasantries, hedging
Short sentences. Active voice.
```

It changes how the agent communicates

Commands provides explicit user control

COMMANDS	
/caveman	activate default full mode
/caveman ultra	maximum compression
/caveman off	deactivate the style
/caveman-commit	concise commit message
/caveman-review	concise review findings
/caveman-compress	compress a memory file
/caveman-stats	session token statistics
/caveman-help	available operations

COMMANDS/CAVEMAN.md

```
---
description: Activate Caveman mode
argument-hint: [lite|full|ultra|off]
---
Activate Caveman mode: $ARGUMENTS
If no level is supplied, use full.
If off, deactivate.
```

Hooks make a prompt-time choice persistent

SessionStart

`caveman-activate.js`

- Decide the mode for this session
- Store the mode state safely
- Inject the rules as hidden context
- Restore the rules after compaction
- Clean old state; offer status line

UserPromptSubmit

`caveman-mode-tracker.js`

- Inspect every user prompt
- Detect /caveman ultra
- Detect phrases like “normal mode”
- Update the session's mode
- Restore mode after focused ops

```
"hooks": {
  "SessionStart":
    ["node caveman-activate.js"],
  "UserPromptSubmit":
    ["node caveman-mode-tracker.js"]
}
```

SessionStart initializes behaviour.

UserPromptSubmit detects changes mid-conversation.

MCP exposes compression as callable tools

FIVE MCP TOOLS	
`caveman_compress`	compress + recovery handle
`caveman_retrieve`	byte-exact original
`caveman_stats`	compression statistics
`caveman_toon_encode`	JSON to TOON
`caveman_toon_decode`	TOON back to JSON

MCP CONFIGURATION

```
{"mcpServers": {
  "caveman": {
    "command": "npx",
    "args": ["-y","caveman-mcp"]
```

EXAMPLE TOOL RESULT

```
"compressed": "3 tests fail.
   Root error: timeout.",
"ratio": 0.72,
"recovery_handle": "ccr_abc123"
```

The proxy compresses what the model reads

TYPE-AWARE COMPRESSION — WHAT IS KEPT	
Logs	errors, stack traces, head + tail
Code	imports, signatures, types, bodies
Diffs	file + hunk headers, changed lines
JSON	keys, structure, error subtrees
Search	high-value matches, diagnostics
HTML / text	headings, relevant sections

PROXY vs MCP

- Intercepts every request; MCP runs only when called
- Between agent and provider; MCP sits beside it
- Proxy changes context; MCP cannot

Hooks intercepts requests — but they cannot *transparently* rewrite request to LLM. It needs a proxy.
If you write your own harness, then yes.

How Capabilities are Used in Caveman

PACKAGE	Caveman plugin — one installation and update unit			
MECHANISMS	**Skills**	**Commands**	**Hooks**	**MCP server**
STATE + ENGINE	Session mode state		Compression engine	
INTERCEPT	Local proxy — between the agent and the model provider			

Skills concise styles, focused workflows

Commands select modes, request operations

Plugin packages the Claude Code integration

Hooks activate, track, reinforce the mode

MCP compression, retrieval, stats, encoding

Local proxy intercepts provider requests

How do you finish the assignments in two hours?

1 · Show up → 2 · Stay awake here → Done before lunch

Become a 100x Engineer With Multiple Long-Running Agents

Parallel Agents by Hand
Subagent
Agent team
Dynamic Workflow

Agents fan out in parallel. So does the invoice.

Three Axes of Scaling

Parallel agents — breadth

- Run multiple independent tasks, each in an isolated working tree
- Buys **throughput**: wall-clock time, not answer quality

Subagents — breadth and depth

- Delegate a bounded task to a fresh context; get back a summary
- Buys **throughput and focus**: parent never sees the intermediate mess

Long-running agent — duration

- Carry a larger job across many steps
- Buys **scope**: more work with less intervention

Agent team and dynamic workflow combine them

Parallel Agents by Hand

One task per session, each in its own **git worktree** — a separate working directory and branch that shares the repository's history.

`claude -w feature-payments`

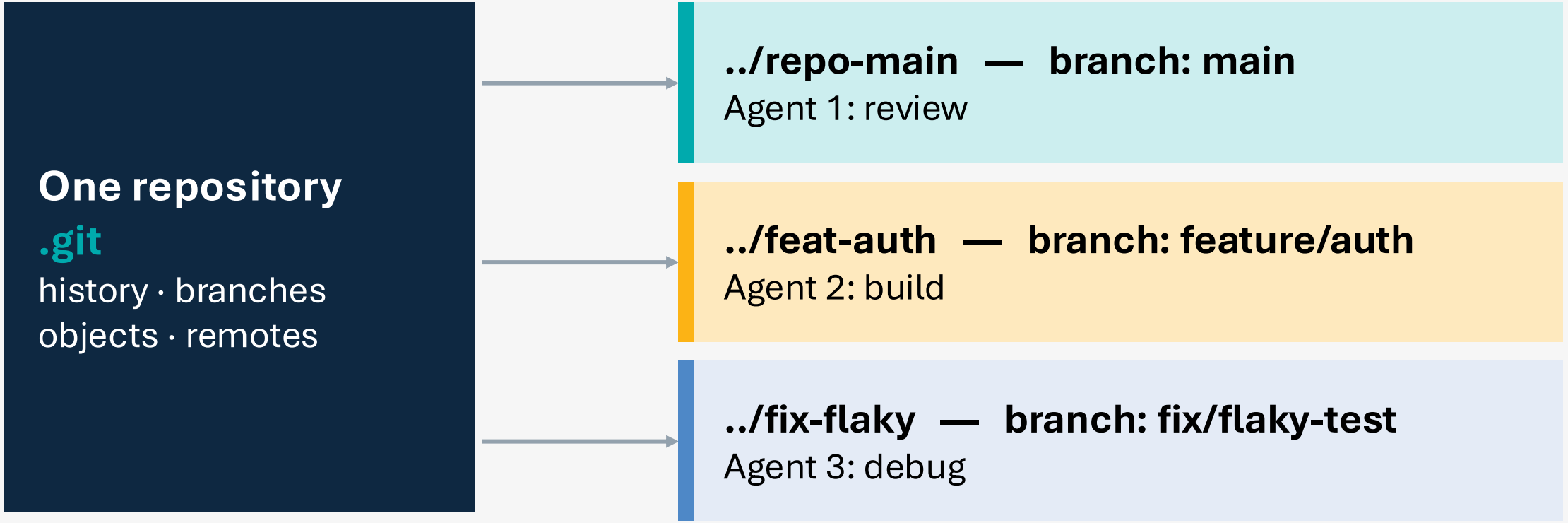
Terminal 1 builds the feature on its own branch

`claude -w bugfix-auth`

Terminal 2 fixes the bug, untouched by the first

Edits in one session never touch the files in another
No stashing, no second clone. In the desktop app every new session gets a worktree automatically.

Git Worktree: One Repo, Many Working Directories



One shared history. Three checkouts. No branch switching, no stashing.

Where Hand-Run Parallelism Stops

A few sessions is a productivity win. Twenty needs a clone of yourself.

You hold the plan — Every hand-off and next step lives in your head

Sessions cannot see each other — Findings move only if you carry them across

You reconcile the branches — Merge order and conflicts are yours to sort out

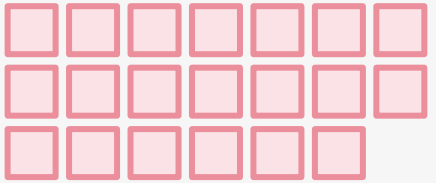
It does not repeat — Nothing about the run is written down to rerun

3 sessions

You can hold the plan

you

20 sessions



You need a clone

you + a clone

Subagents: Why?

A specialized assistant with its own context window, tools and permissions.

Preserve context
Logs, search results and file dumps stay out of the main conversation.

Enforce constraints
Limit which tools a subagent may use, and which model runs it.

Reuse anywhere
Focused prompts for a domain, shared across projects and teams.

Parallelization
Independent subtasks run concurrently, so the slowest one sets the total time.
style-checker, security-scanner and test-coverage at once

Control costs
Route tasks to faster, cheaper models like Haiku.
model: haiku in the subagent's frontmatter

code.claude.com/docs/en/sub-agents

Built-In Subagents

Claude delegates to these automatically

Explore
Read-only search of the codebase. Write and Edit denied; thoroughness set per call.

Plan
Research during plan mode, so exploration stays out of the main window.

General-purpose
Complex multi-step work that needs both exploration and action.

Explore and Plan skip CLAUDE.md and git status to stay fast and cheap.

Defining a Subagent

A Markdown file with YAML frontmatter; the body becomes the system prompt.

```
.claude/agents/code-reviewer.md
---
name: code-reviewer
description: Reviews code for quality
tools: Read, Glob, Grep
model: sonnet
---
You are a code reviewer. Analyze the
code and give actionable feedback.
```

`.claude/agents/`

Project scope — check into version control so the team shares it.

`~/.claude/agents/`

User scope — available across all of your projects.

Defining Nested Subagent

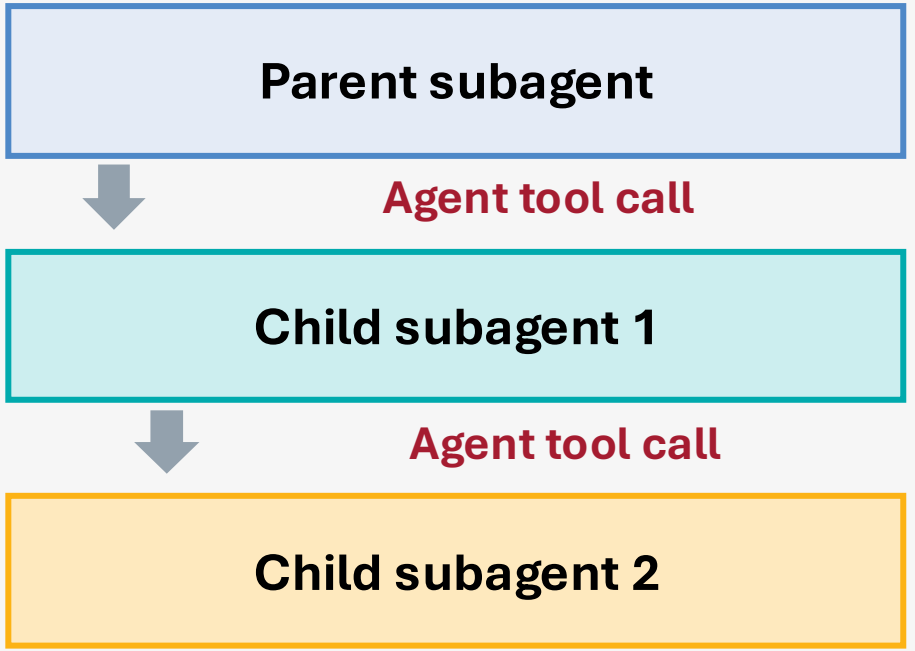
Sub-agent frontmatter

```
---
name: my-orchestrator
tools: Read, Grep, Agent
model: claude-sonnet-5-0
---
```

Up to 5 levels of nesting are allowed

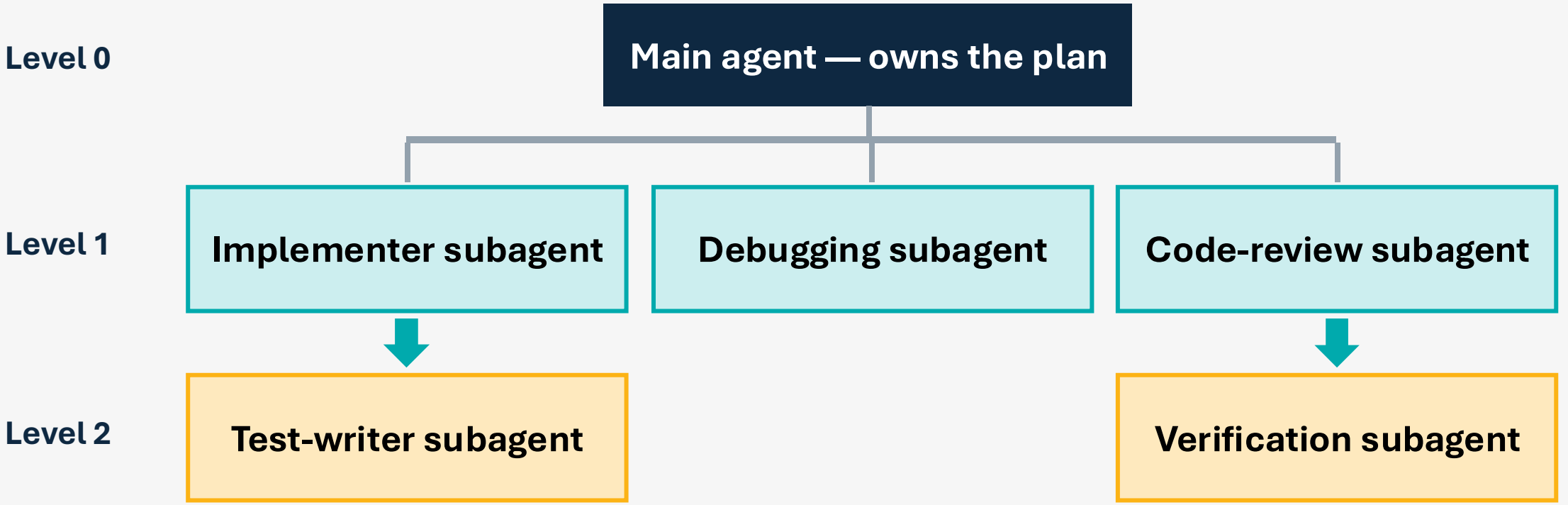
2 to 3 levels is the sweet spot

The call itself



Agent must appear in tools before a subagent can spawn children

Multi-Level Subagents: A Common Pattern



Every level starts with fresh context and returns only a summary

Work With Subagents

Automatic delegation happens on its own — escalate explicitly when it doesn't.

1 · Natural language

Name the subagent in your prompt — Claude decides whether to delegate.

```
Use the test-runner
subagent to fix tests
```

Scope: ONE-OFF

2 · @-mention

Pick the subagent and it is guaranteed to run.

```
@agent-code-reviewer
look at auth changes
```

Scope: ONE TASK

3 · --agent flag

The whole session adopts its prompt, tools and model.

```
claude --agent
code-reviewer
```

Scope: WHOLE SESSION

Increasing control over which subagent runs

Subagent, or Main Conversation?

A subagent starts fresh: it sees no history, and pays a latency cost to gather context.

Keep in main conversation

- Iterative back-and-forth
- Phases that share context
- Quick, targeted changes
- Latency matters

Delegate to a subagent

- Verbose output you won't reuse
- Tool or permission restrictions
- Self-contained work returning a summary
- Parallel, independent work
- Simple work can delegate to small model

Want a reusable workflow in the main context instead? Use a Skill.

Agent Teams

Several Claude Code sessions working together: one lead, several teammates.

Review
Teammates probe different angles, then challenge each other's findings.

Independent modules
Each teammate owns a separate piece without stepping on the others.

Debugging
Competing hypotheses tested in parallel converge faster.

Cross-layer
Frontend, backend and tests, each owned by a teammate.

Not free: coordination overhead and token cost scale with the team.
Sequential tasks, same-file edits or heavy dependencies — use one session or subagents.

Experimental — off by default.

code.claude.com/docs/en/agent-teams

How a Team Is Wired

Four parts, and teammates talk to each other directly.

Team lead
The main session: spawns teammates, coordinates work, synthesizes results.

Teammates
Separate Claude Code instances, each with its own context window.

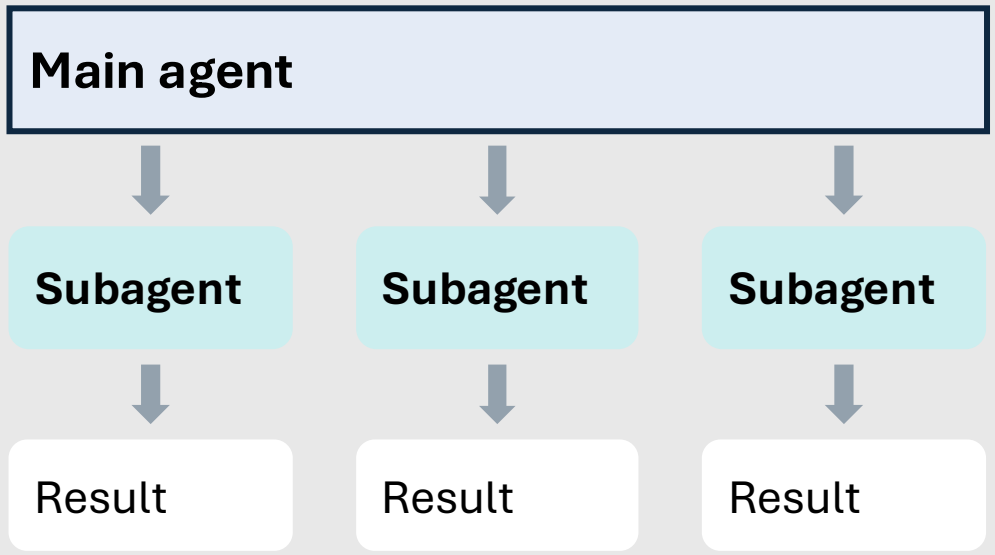
Task list
Shared work items that teammates claim and mark complete.

Mailbox
A JSON inbox per agent.

Teammates do not inherit your conversation.
They load CLAUDE.md, and skills automatically and put the task detail in the prompt.

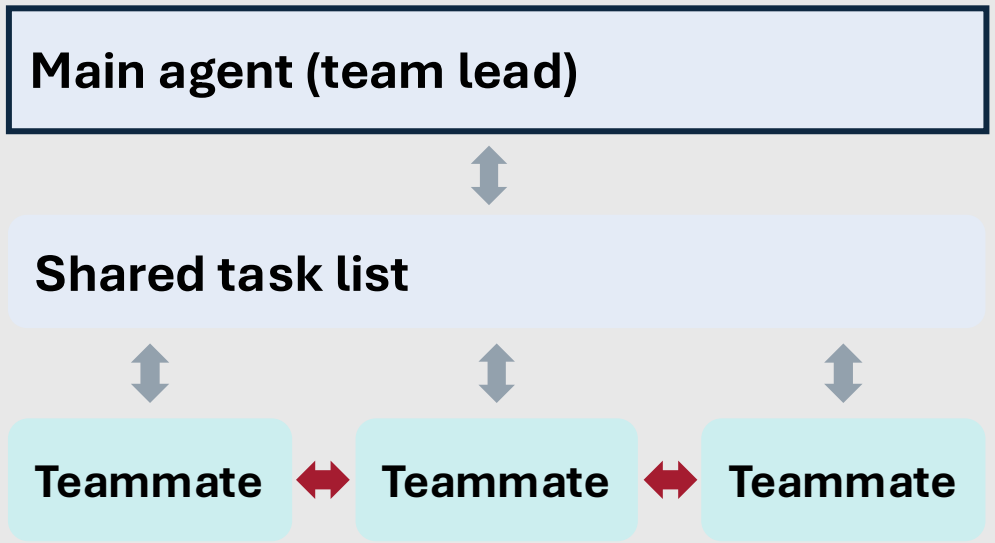
Agent Team vs Subagent

Subagents



Results report back to the main agent

Agent teams



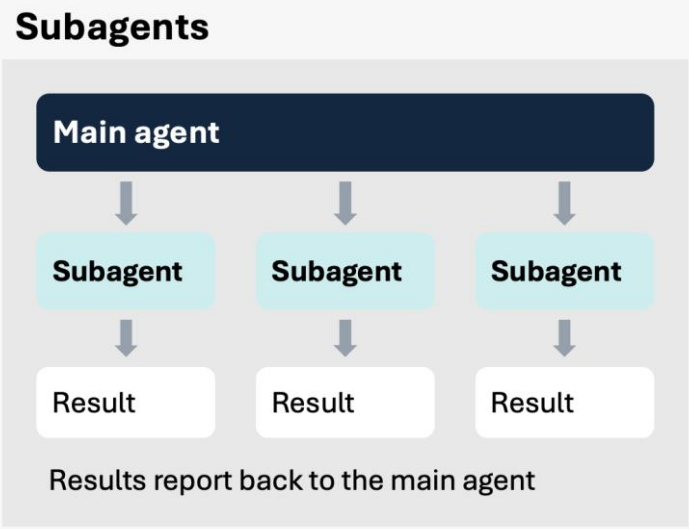
Claim tasks and message each other

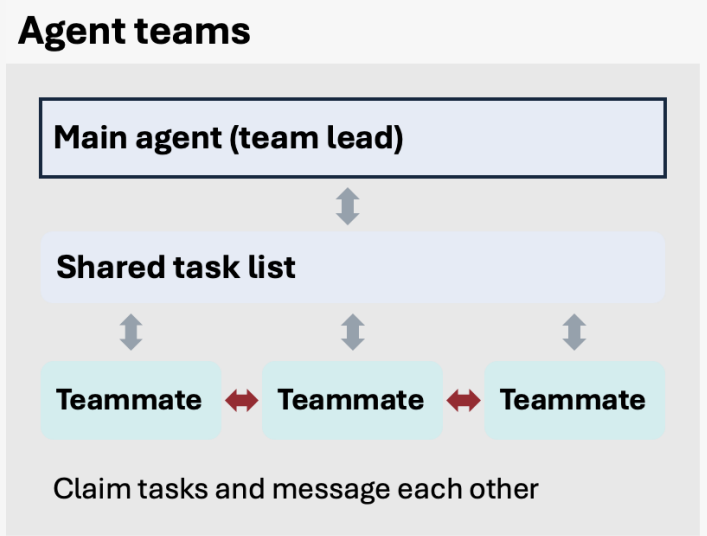
Subagents report back to the lead. Teammates share a task list and talk to each other.

Subagents vs Agent Teams

	Subagents	Agent teams

Teams cost more — use them only when the work needs discussion



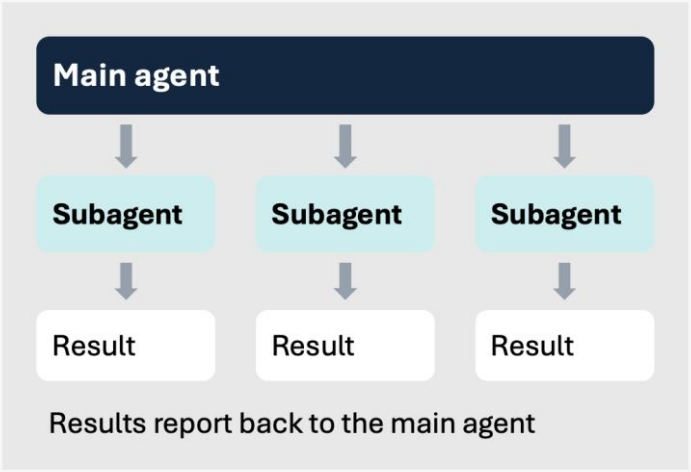


Subagents vs Agent Teams

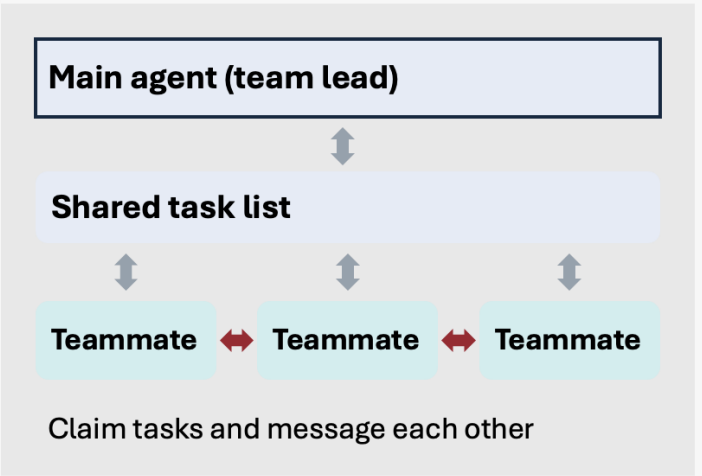
	Subagents	Agent teams
Context	Own window, result to caller	Own window, independent
Comm.	Back to the caller only	Teammate to teammate
Coordination	Lead manages the work	Shared task list
Best for	Result-only tasks	Work needs discussion
Token cost	Lower	Higher

Teams cost more — use them only when the work needs discussion

Subagents



Agent teams



Dynamic Workflow

A JavaScript *script* that orchestrates many subagents in the background.

Use it when a task needs more agents than one conversation can coordinate

Codebase-wide bug sweep	**A 500-file migration**	**Research cross-checked**	**A plan from several angles**

Two ways to start one

`ultracode`

Type it in your prompt

`/effort ultracode`

Let Claude decide for you

Both routes spawn many subagents — plan, design, implement, review.

Long-Running Agents: /goal

Hand over a verifiable end state; Claude keeps working until it holds.

```
/goal "npm run lint exits with zero errors"
```

Define done

A condition Claude can test: tests pass, zero lint errors.

It self-checks

An evaluator rechecks the condition after each attempt, then sends Claude back in.

It stops itself

Ends when the goal is met, or when --max-tries runs out.

Verifiable!

"all integration tests pass" works, "refactor the code" does not.

Long-Running Agents: /loop

Re-run a prompt on a schedule; it keeps going until you stop it.

```
/loop 5m "check CI, fix whatever broke"
```

A trigger, not a test

You hand off the interval; the same prompt runs again each time.

You end it

It iterates until you press Esc while it waits between runs.

Its own default

Put a default prompt in .claude/loop.md

/goal stops when the condition holds. /loop stops when you ask it.

Summary

Four ways to amplify with multiple agents — and two ways to keep one running.

More agents at once

Parallel agents, by hand — you reconcile them yourself

Subagents — Claude delegates; results come back to it

Agent teams — a lead supervises peers over a shared list

Workflows — a script holds the plan, not the chat

One agent, for longer

`/goal`
Runs until a verifiable end state holds

`/loop`
Re-runs on a schedule until you stop it

OPTIONAL — extra reading, not required

Agent Team Example: Sixteen Agents Built a C Compiler

A Rust C compiler that builds the Linux kernel — 16 agents, ~2,000 sessions, two weeks

Scale

- 16 agents working in parallel
- ~2,000 sessions over two weeks
- 2B input / 140M output tokens
- Just under $20,000 in API cost

Result

- ~100,000 lines of code
- Linux 6.9 on x86, ARM and RISC-V
- QEMU, FFmpeg, Postgres, Redis, Doom
- ~99% pass rate on major test suites

https://www.anthropic.com/engineering/building-c-compiler

OPTIONAL — extra reading, not required

How the Sixteen Agents Coordinated

Plain infrastructure — containers, Git and text files — did the orchestration

- Each agent in an isolated container and Git checkout
- Tasks claimed through simple lock files
- Git as the coordination and integration layer
- Fresh sessions read progress docs to regain context
- Specialist agents for docs, performance, quality, dedup
- Automated tests gave agents a progress signal

OPTIONAL — extra reading, not required

Git as the Coordination Layer

No chat, no manager agent — the agents coordinated through shared Git state

Read the repo
clone, scan docs and logs

→

Claim a task
lock file in current_tasks/

→

Implement and test
in its own container

→

Merge, push, unlock
pull everyone else first

The loop repeats — agents pick the next obvious task themselves; Git settles duplicate claims

Conflicts settle themselves

- Same task claimed twice? One agent moves on

Artifacts carry the context

- READMEs, progress docs, CI failures brief new sessions

OPTIONAL — extra reading, not required

Successful but Not Fully

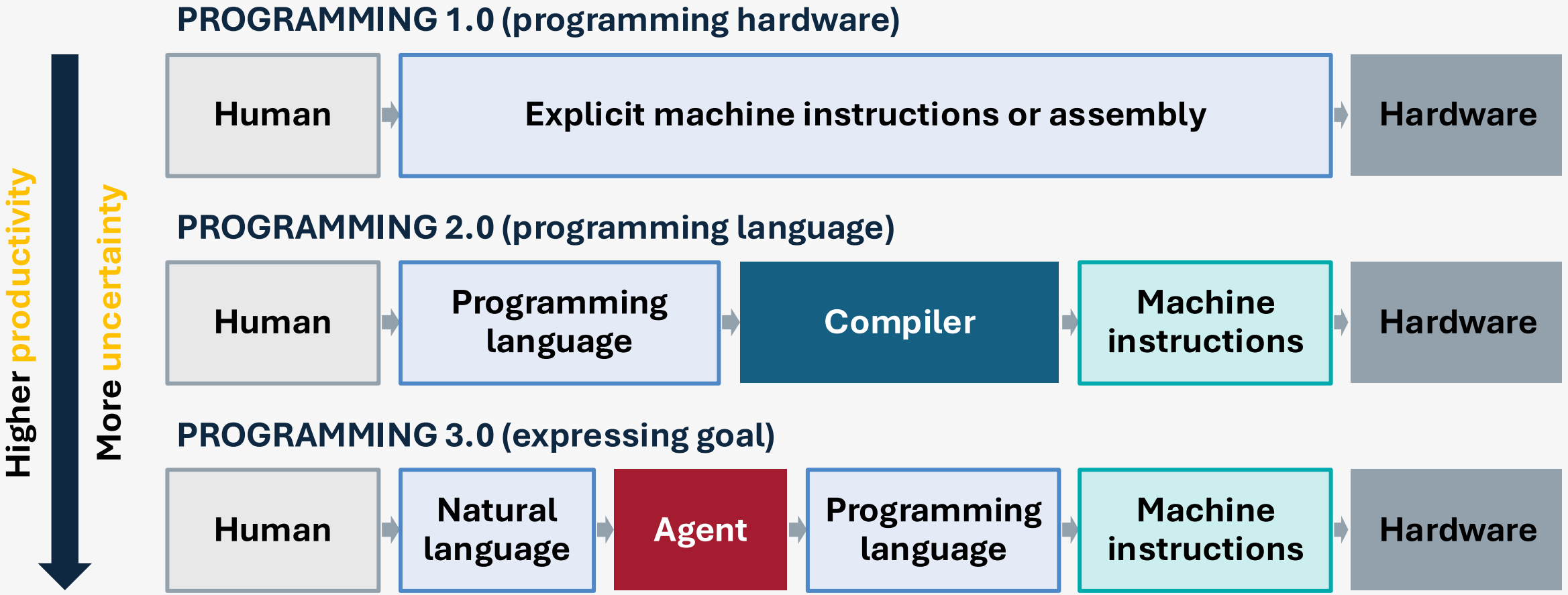
Agent is not yet perfect, more work ahead for you!

- Merge conflicts were frequent
- Parallelism stopped helping once every agent hit the same kernel failure
- Still leaned on GCC for part of 16-bit x86
- Generated code was less efficient than GCC output

Programming / Software 3.0

This is a term coined by me, different from Andrej's definition (next slide)

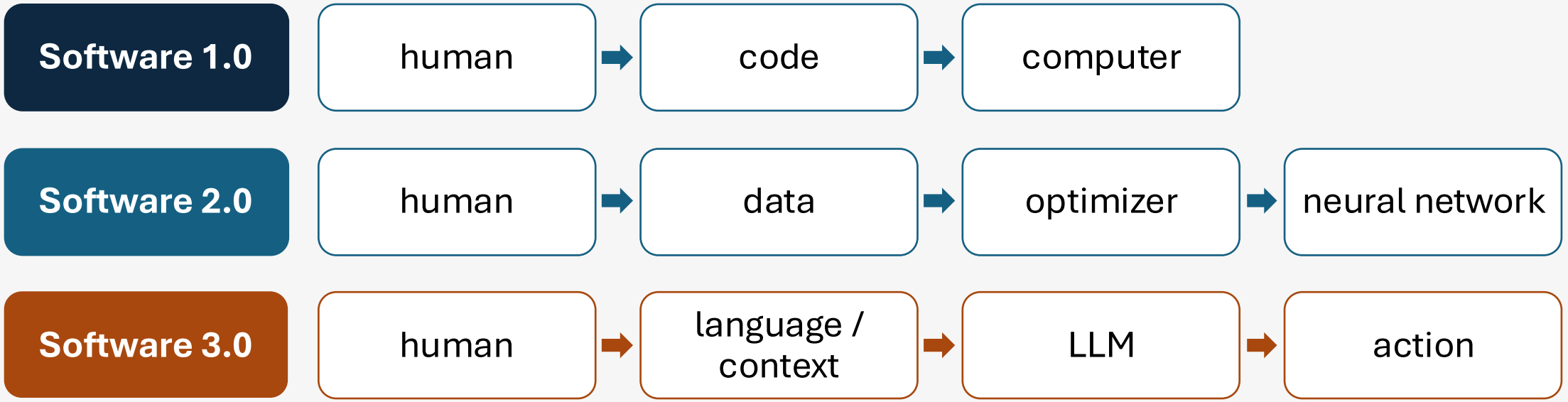
Programming 3.0



Software 3.0

Andrej Karpathy

Paradigm	What is the program?	How to create it?
1.0	source code	write Python/C++
2.0	neural-network weights	collect data + optimize
3.0	LLM prompts	natural language



> Knowing where to draw the boundary between 1.0, 2.0 and 3.0 is an important engineering skill.

Uncertainty in Programming 3.0

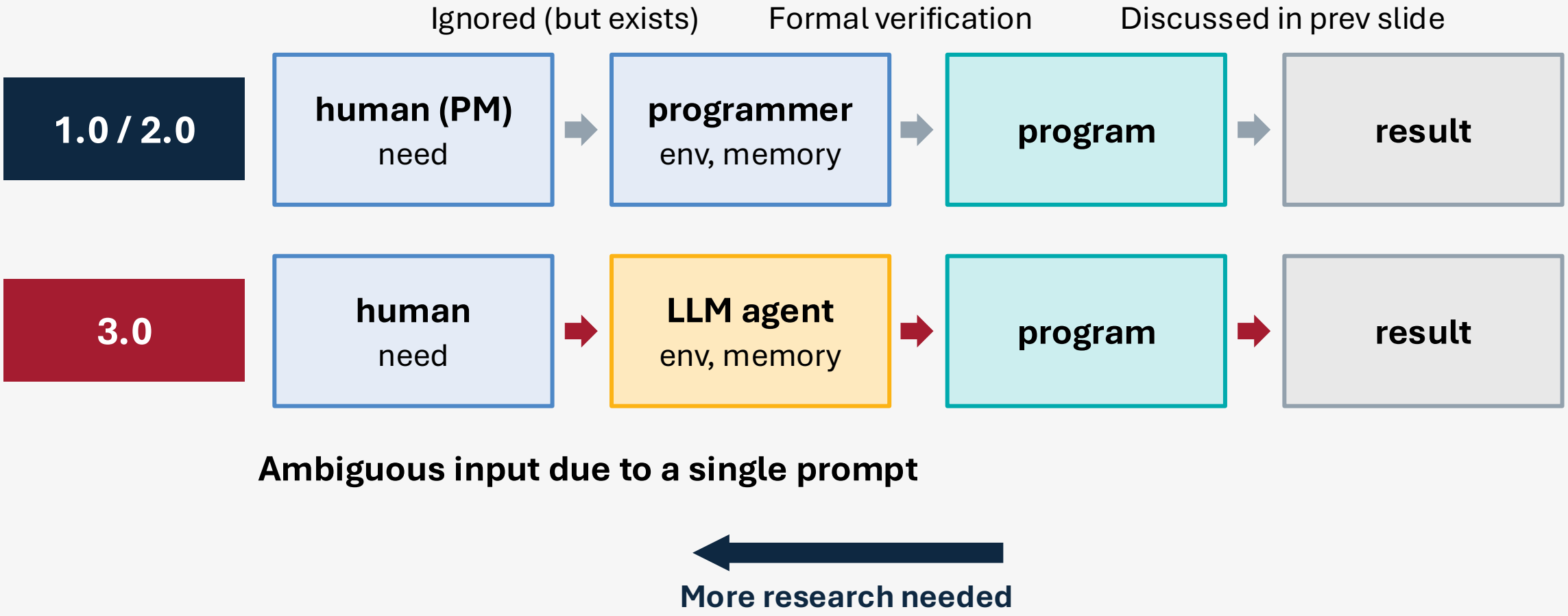
Uncertainty becomes the core theme,
and you should learn to live with it!

Is uncertainty new?

The same source code != same output happened a long time ago

Uncertainty source	Programming 2.0	How 2.0 solves	Programming 3.0 adds
Hardware	architecture: 32-bit vs 64-bit, endianness	fixed-width types, cross-compilation, virtualization	**GPU compute** nondeterministic kernels, floating-point variance
Environment	concurrency, unreliable networks	memory consistency model, idempotent APIs	**Interaction at codegen** it touches the environment as it writes
Software	compiler versions, dependency drift	comprehensive test before upgrade	**Compiler, model** different kernels, sampling

A bit more uncertainty: from intent to result



Live with Uncertainty

Natural language	**semantic uncertainty**	
Retrieval / context	**retrieval drift** index updates, chunking, context truncation	
Model generation	**probabilistic variability**	1. sampling 2. software and hardware non-determinism
Agent/tool loop	**trajectory / environment nondeterminism**	tool failure environment change
Generated program	**usually deterministic once generated** unless it calls models or tools at runtime	

Best Practices for Programming / Software 3.0

1 · DEFINE	2 · CONTROL	3 · VERIFY & GOVERN
State assumptions up front Have the model confirm anything ambiguous before it executes.	**Separate proposal from execution** Agents propose plans; validate before anything commits.	**Tighten the generate-verify loop** Cheap, frequent checks beat one expensive review at the end.
Specify outcomes, not prompts Give requirements, invariants, constraints and acceptance tests.	**Manage context explicitly** Own memory, provenance and retrieval; expire stale information.	**Measure one-level deeper** Create tests, measure multiple metrics, synthetic workloads that you can reason about

Do not expect the agent to be deterministic; make the surrounding system dependable.

Match autonomy to risk: the more autonomy the higher uncertainty and risk

Evidence From the Field

Evidence from Codex (June 2026)

OpenAI employee usage (September 2026)

From Conversation to Delegation

Conversational AI

Human asks a question

↓

Model answers
Human stays in the loop

↓

Human executes the work

The human is the executor.

Agentic AI (Codex)

Human delegates a task

↓

Agent plans, calls tools, edits files, runs commands

↓

Artifact or action delivered

The human reviews and integrates.

Used Codex in last 28 days

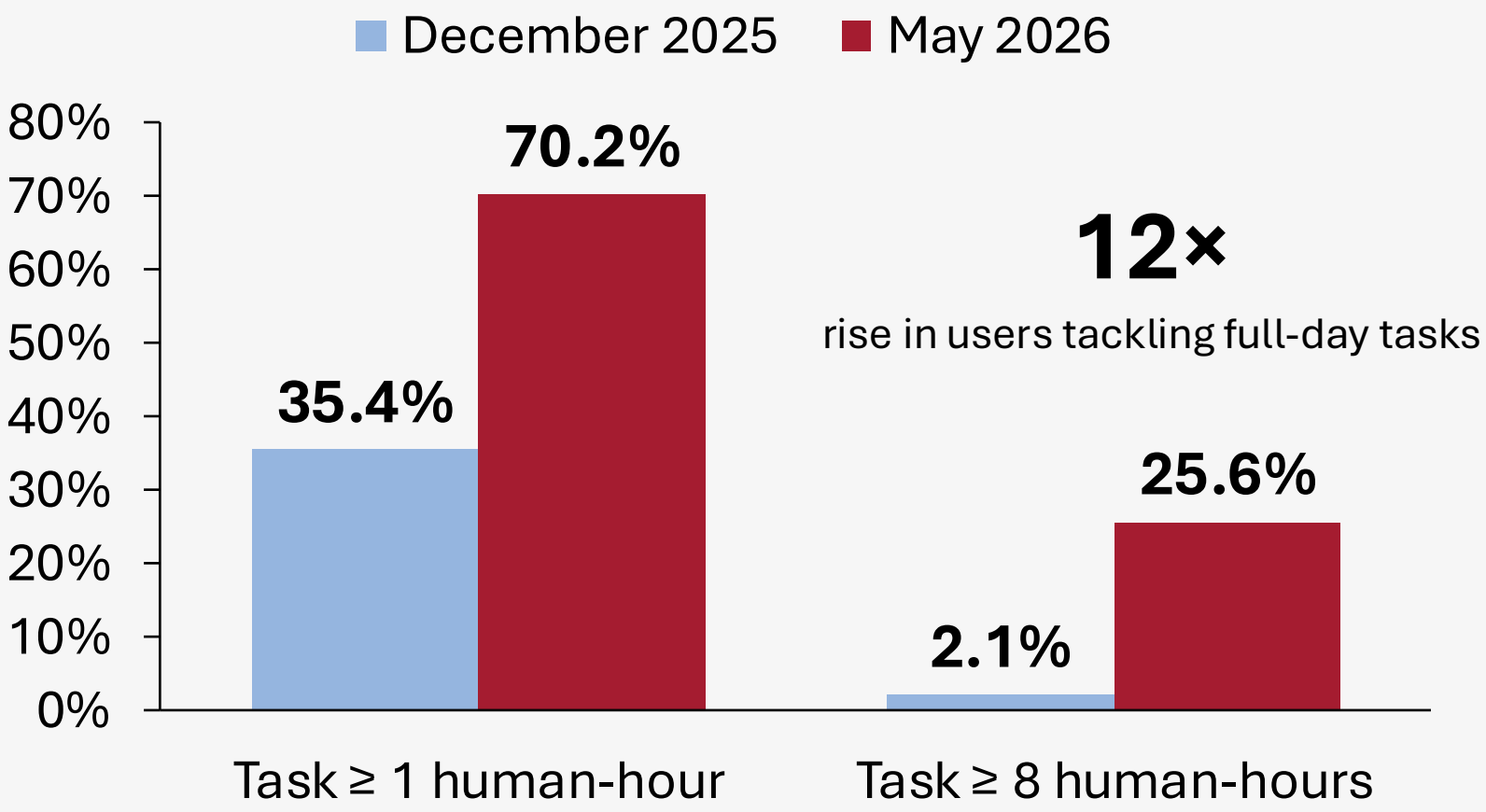
Individual
< 1%

Organizational
17.3%

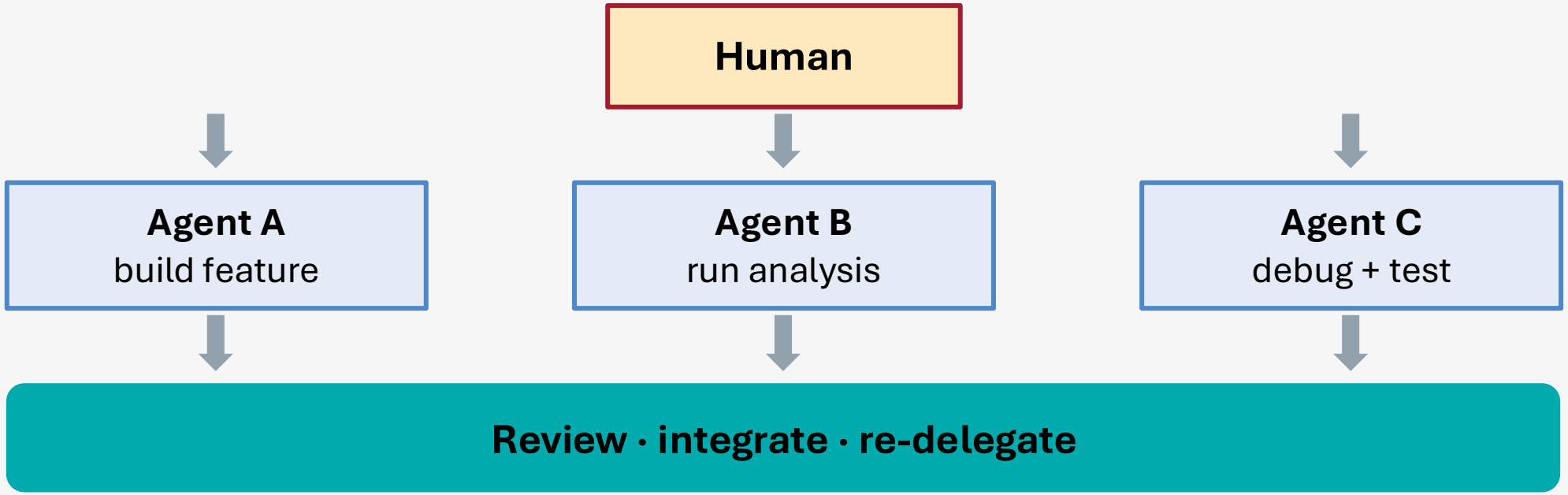
OpenAI workers
nearly all

Active users grew more than 5× in the first half of 2026.

Delegated Tasks Are Getting Much Bigger



One Human, Many Agents



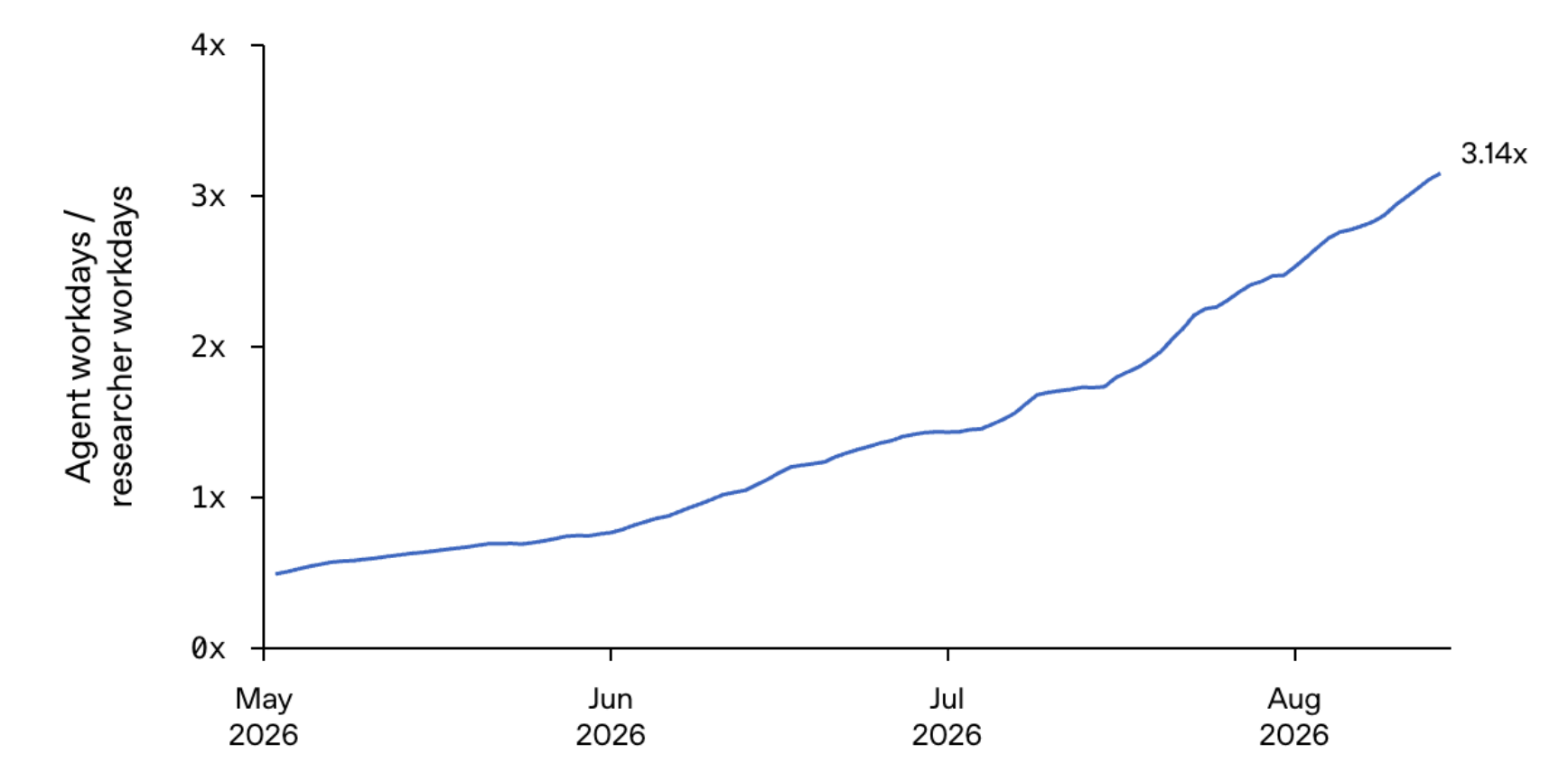
28.6%
of OpenAI users ran 5+ agents at once in a week

~2.5 agent-hours
per day for the median OpenAI employee

~71 agent-hours
per day at the 99th percentile (+88% since April)

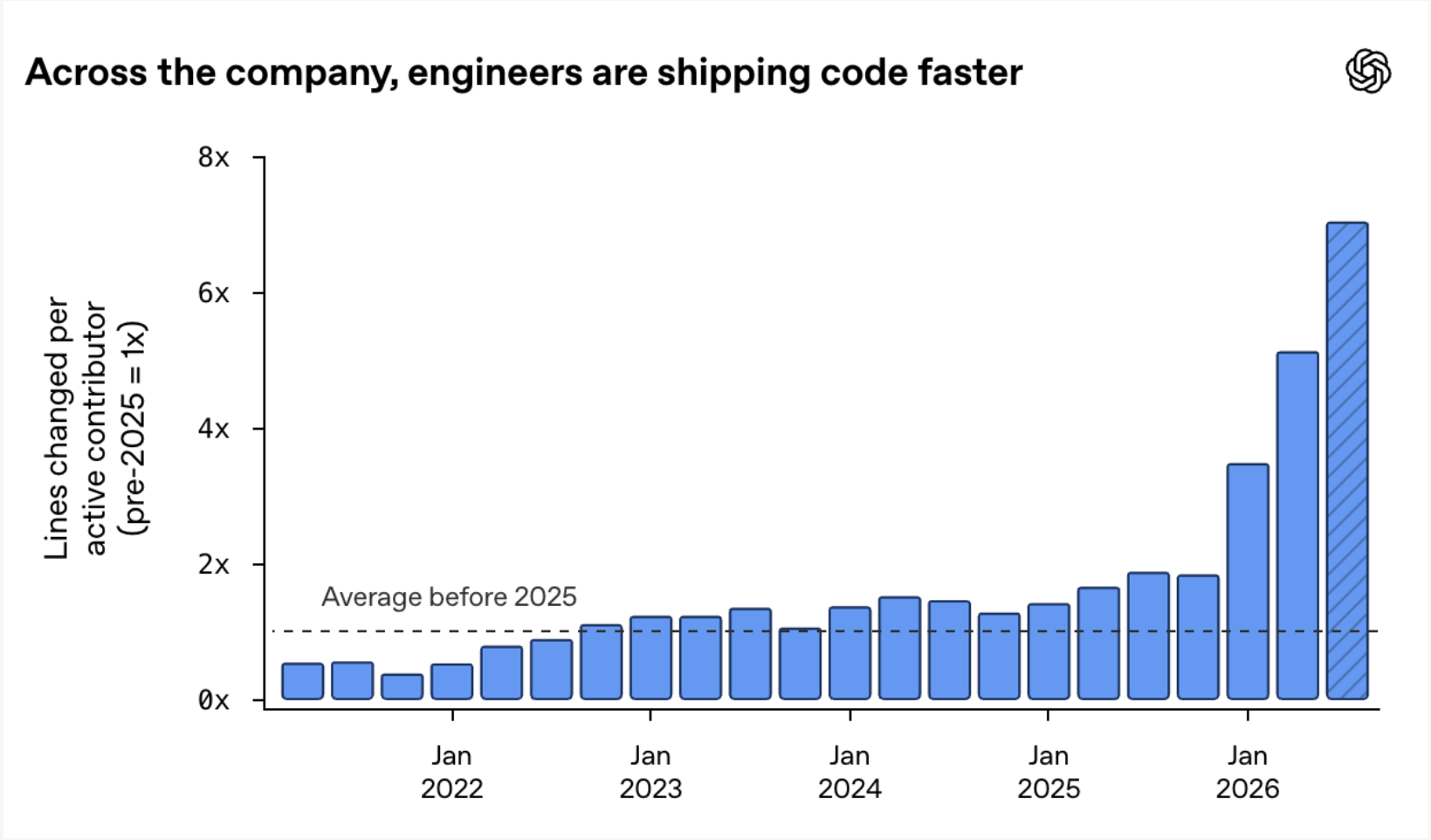
The human role shifts to planner, delegator, monitor, reviewer, integrator.

Agentic Workdays Now Far Exceed Human Researchers

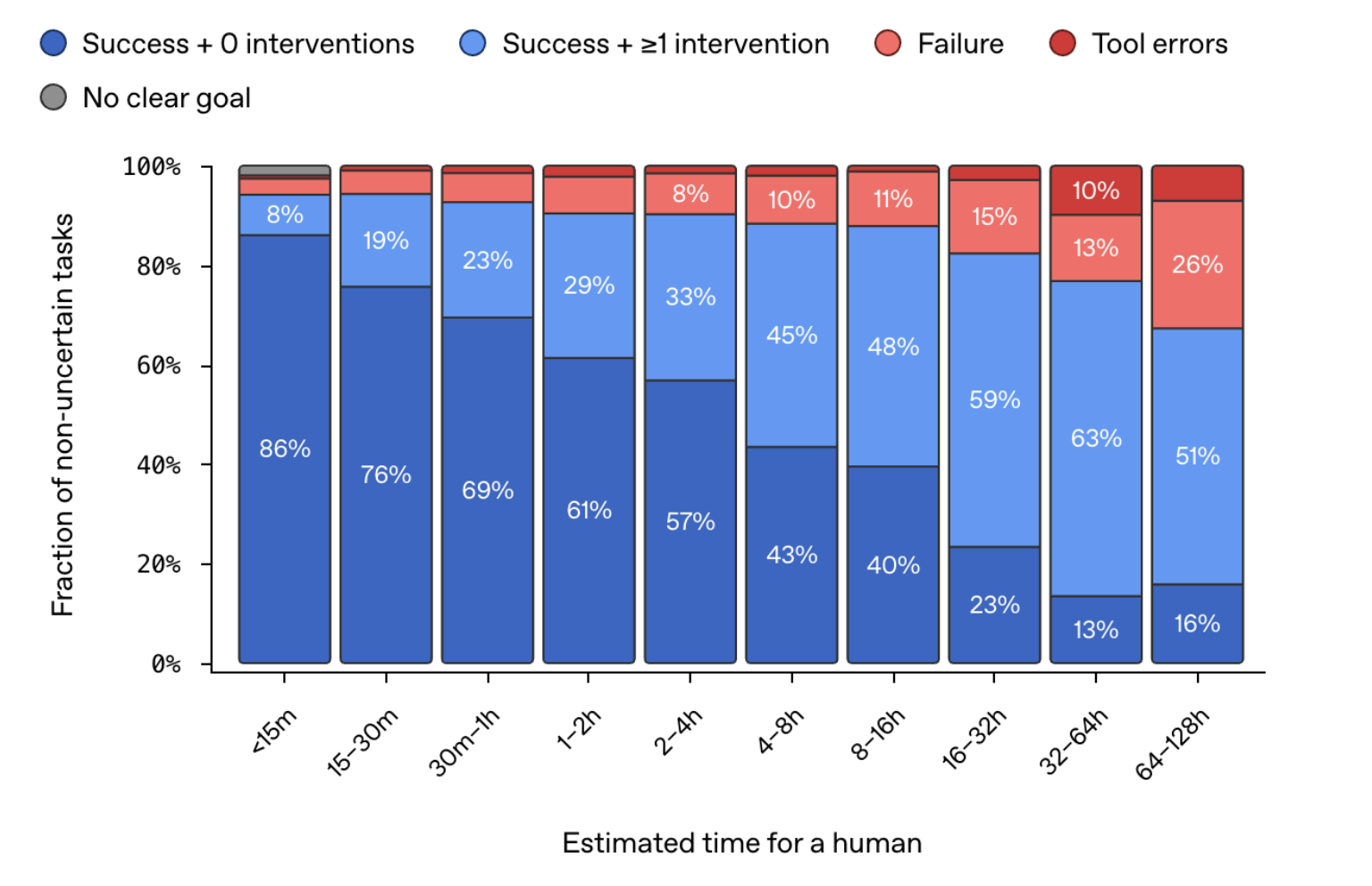


Before June 2026 agent runtime trailed human labor, now agent do 3.14x more work than human. Most researchers have 4+ concurrent workflows (not shown).

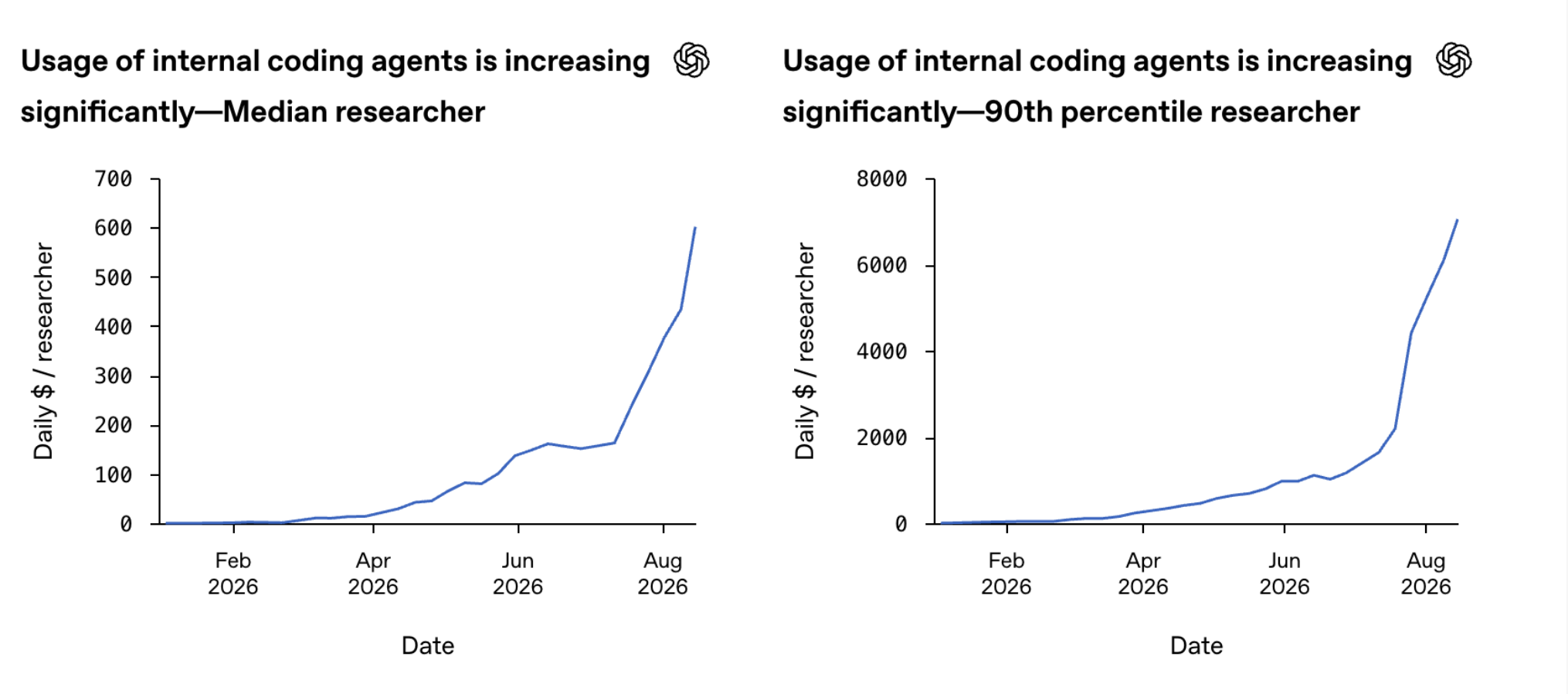
Engineers Are Shipping Almost 8x Faster



Agents Solve Increasingly More Complex Tasks But Human Intervention is Still Needed



Coding Agents Are Reshaping Daily Work for OpenAI Researchers



September 2026

The Delegation Ladder

1. **Chatbot — AI answers; the human does the work**
2. **Assistant — human and AI work on the task together**
3. **Agent — human delegates a whole unit of work**
4. **Multi-agent worker — human coordinates several agents**
5. **Systematized workflows — skills and agents run processes**

OpenAI's workforce sits near the bottom.

Next Class

- Agent design I
- Find TF to get your participation score

Optional Slides

Beyond Claude Code

Model labs model + harness in one	**Platform CLIs** inside a dev stack	**Open source** bring your own key	**Minimal cores** small and auditable
Claude Code	Copilot CLI	OpenCode	Deepseek Harness
Codex CLI	Cursor CLI	Crush · Goose	Pi · oh-my-pi (Omp)
Antigravity CLI	Amp	Aider · Cline	mini-SWE-agent
Grok Build	Auggie	Kilo · Continue	gptme
Mistral Vibe	Droid	OpenHands	Open Interpreter
Kimi Code	Junie CLI	Reasonix	
Qwen Code	Qoder	Codebuff	
	CodeBuddy		

State of CLI Coding Agents, Mid-2026 (July 2026)

How They Differ: Five Axes That Matter

Open vs. closed	**Open:** Codex CLI, OpenCode, Goose, Cline, Pi **Closed:** Claude Code, Copilot CLI, Cursor, Droid, Junie
Locked vs. BYOK	**One vendor:** Claude Code, Antigravity CLI, Grok Build **Any endpoint:** OpenCode 75+ providers, Kilo 500+ models
Sandbox vs. prompt	**OS sandbox:** Codex CLI, Copilot CLI, OpenHands containers **Permission prompts:** Claude Code and most others
Local vs. cloud	**Local-first, no telemetry:** Goose, Nanocoder, gptme **Cloud handoff:** Copilot CLI, Codex, Mistral Vibe remote agents
Maximal vs. minimal	**Everything included:** Omp (widest feature set), Claude Code **Four tools only:** Pi, mini-SWE-agent (~100 lines of Python)

State of CLI Coding Agents, Mid-2026 (July 2026)