Introduction to Agentic Software Engineering

Oscar Chaparro

Agenda

1. What is agentic AI — the brief picture
2. The agentic harness
3. Tool landscape: modes and autonomy
4. What you can and cannot delegate
5. Practical patterns
6. Validation and responsibility

The shift

Autocomplete

Suggests completions as you type. You accept or reject.

→

Chatbot

Answers questions, generates code on demand. One exchange.

→

Agent

Plans, calls tools, observes results, iterates until the task is done.

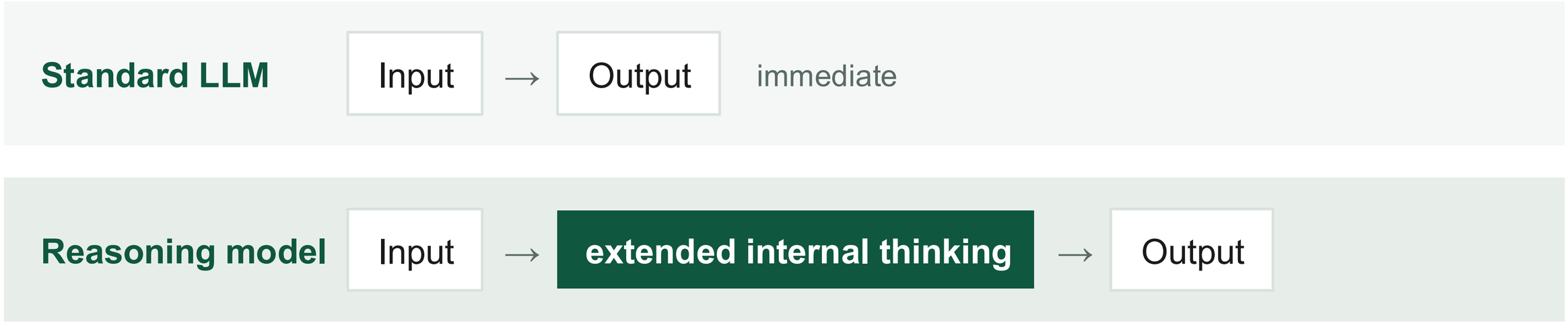
What is agentic AI — the brief technical picture

How LLMs work

↺ the new token is appended to the context, and the whole thing repeats

- A language model predicts the most likely next token, given everything before it.
- Training on massive text → emergent reasoning as a side effect of scale.
- **Key implication: output quality depends heavily on what is in the input.**

Reasoning models



- The "thinking" is real computation at inference time, not post-processing.
- Tradeoff: slower and more expensive, but more accurate on complex multi-step problems.

The agent leap

LLM + ↺ **loop** + **tools**

files · terminal · APIs · search · subagents

LLM alone: answers once, forgets, cannot act on the world.

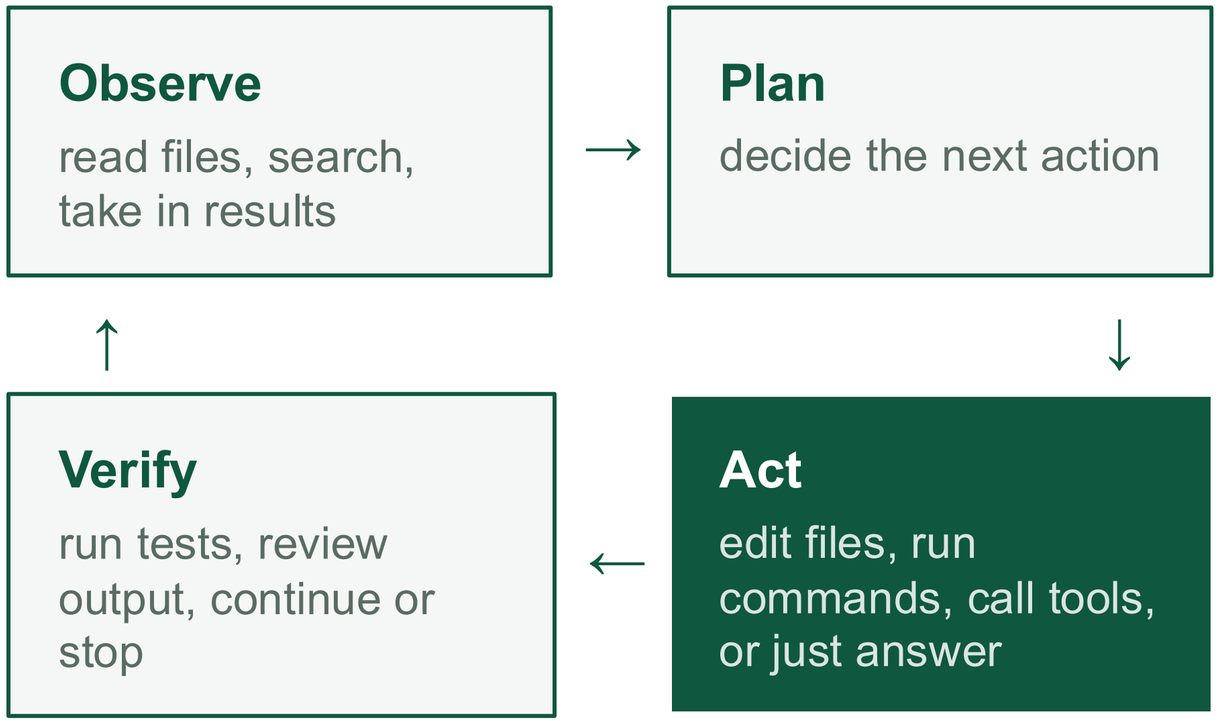
Loop: the model observes results and acts again.

Tools: read/write files, run shell commands, call APIs, search, spawn subagents.

The agent *iterates* on a task — it does not just answer.

The agentic harness

The agent loop



TOOLS AVAILABLE AT "ACT"

Read file	Write file
Run shell	Run tests
Search	Call API
Spawn subagent	

YOU can interrupt at any point — steer, add context, ask for another approach

The context window

CONTEXT WINDOW

System prompt
instructions, persona, tools, rules

Conversation history
what has been said and done

Loaded files
code, docs, test output

Tool results
what previous actions returned

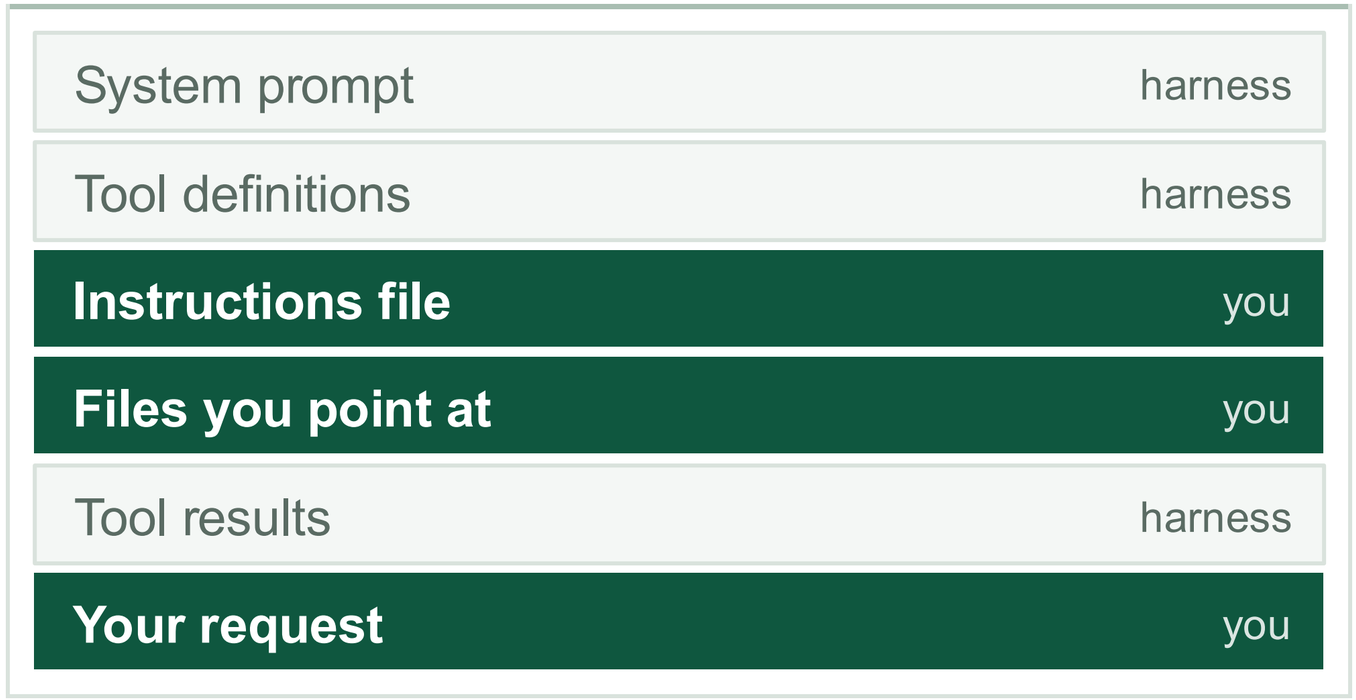
Everything the model can see right now.

Finite. You control what goes in.

Context engineering: deliberately shaping what the agent sees and knows.

"The agent is only as good as its context."

Who engineers the context?



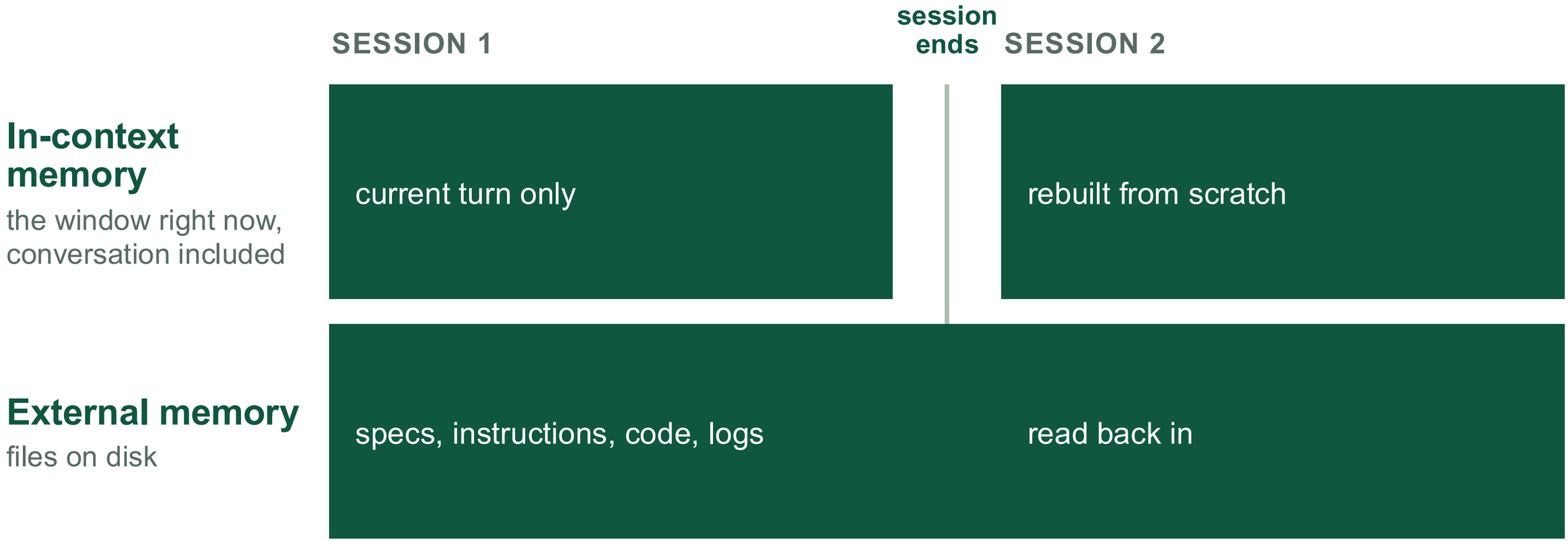
You fill this

The harness fills this

When the window fills, the harness decides what to trim.

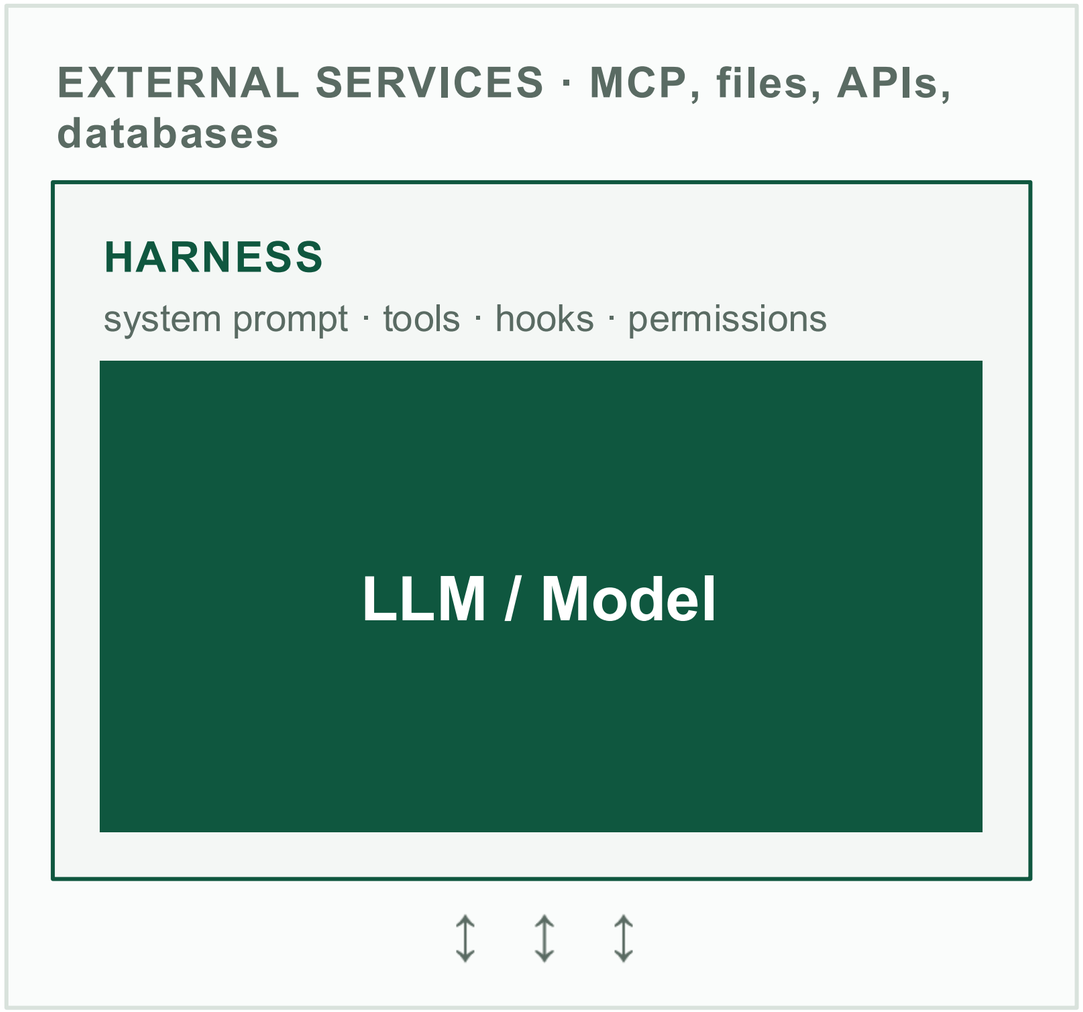
A shared job: the harness assembles the window, you decide what goes in it.

What survives a session



Only what you write deliberately (specs, instructions, planning files) carries *intent* across sessions.

Harness architecture



- System prompt / instructions file
- Tools — file I/O, shell, APIs, search
- Hooks — scripts on lifecycle events
- Permission tiers
- Subagent configurations
- MCP — connects agents to external services

"A harness is largely a delivery mechanism for good context engineering."

Whose harness is it?

Ships with the tool	the loop · tools · compaction · permission system
You adapt it	instructions file · permissions · MCP servers · subagents · what you point it at
Build from scratch	only if the agent is your product

You are not building a harness — you are specializing a general one for your project.

Coding vs. general harnesses

Coding harnesses

- Deep codebase access: read, write, run, test
- Version control integration (git)
- Language-aware context (symbols, imports)
- **Examples:** Claude Code, Cursor Agent, Codex CLI

General harnesses

- Broader tool access (web, files, email, calendar, APIs)
- Less code-specific context
- Can spawn specialized subagents for coding tasks
- **Examples:** Claude (agentic mode), GPT-4o with tools

This course uses coding harnesses for the project.

What makes it a coding harness

GENERAL	CODING
web search, sandbox	read / edit files, grep, shell, git, run tests
everything in the window	search the repo, read only what matters
generic instructions	read before editing, match conventions, minimal diffs
no ground truth	tests, types, build — automatic feedback
chat cannot break anything	approval gates, branches, sandboxing

Same model, same loop — the difference is tools, context policy, and verification.

The tool landscape — interaction modes and autonomy

Interaction modes

Inline / autocomplete

Suggests completions as you type. You accept or reject one token at a time. Always interactive.

Chat

You ask, it generates. One exchange. You drive entirely.

Agentic (multi-step)

Given a task, it plans and executes multiple actions. You review at checkpoints.

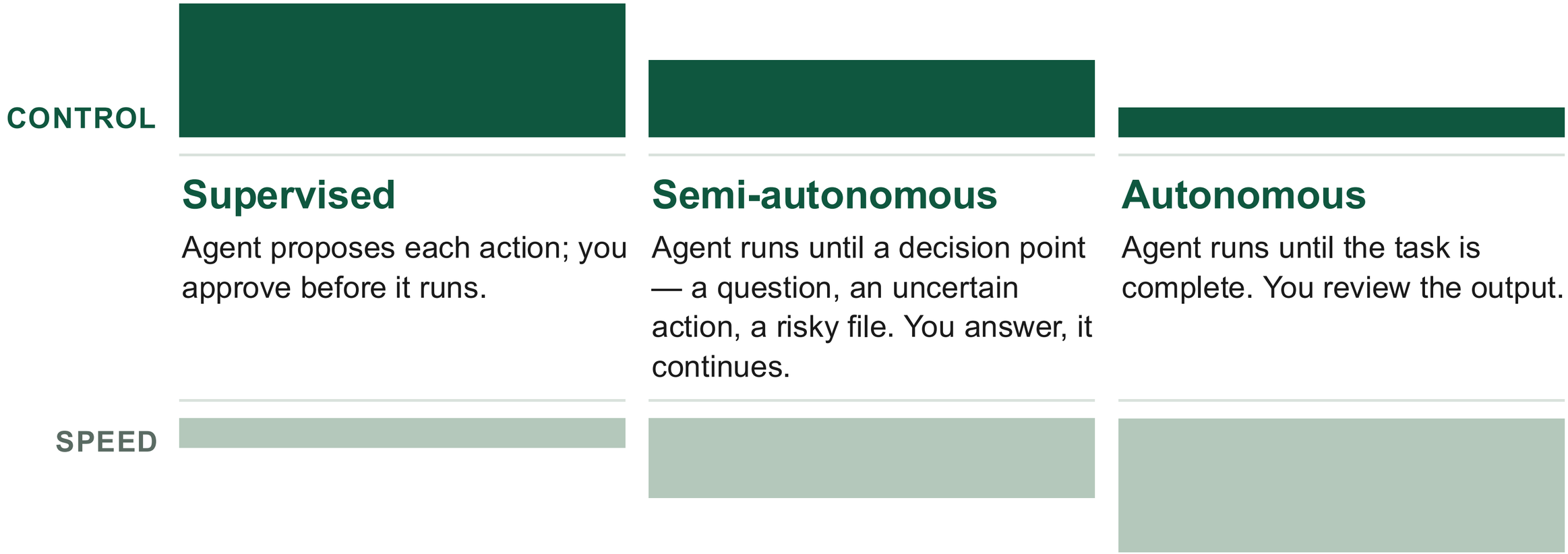
Autonomous / background

Runs without your presence. Returns results when done.

INCREASING AUTONOMY →

These are not mutually exclusive — a session can move between modes.

Autonomy levels



Supervised

Agent proposes each action; you approve before it runs.

Semi-autonomous

Agent runs until a decision point — a question, an uncertain action, a risky file. You answer, it continues.

Autonomous

Agent runs until the task is complete. You review the output.

Right level depends on: task risk, your familiarity with the codebase, and trust built with this agent on this type of task.

Where the agent runs

Surface	Codebase access	Best for
CLI Claude Code, Codex CLI	Full filesystem, shell, git, MCP	Long-running tasks, large context, scriptable workflows
IDE Cursor Agent	Open files, editor state, inline diff	Iterative in-editor changes, tight feedback loop
Web Claude.ai, ChatGPT	Only what you paste in	Questions, exploration — no direct codebase access
Desktop agents Claude computer use, Claude in Chrome	System-level: sees the screen, controls any app	Cross-app automation, GUI tasks — not primary for coding

All of these are converging — the question is not which tool, but which workflow.

What you can and cannot delegate

The delegation map

✓ Wrong answers surface fast

delegate

- Boilerplate and repetitive code
- Test scaffolding you review
- Refactoring with clear rules
- API docs and code explanation
- Searching across a large codebase
- Implementing a well-specified, scoped task

⚠ Wrong answers surface late

keep or review

- Architecture decisions (long-term tradeoffs)
- Requirements clarification (resolving ambiguity)
- Security-critical code — never unreviewed
- Business logic with implicit or unstated rules
- Verifying whether the agent's approach is actually correct

Technical limitations

Hallucinations

Confident, plausible, wrong output. Obscure libraries, your internal conventions, and recent API changes fail most.

Context drift

Over long sessions, the agent loses track of the original goal and starts optimizing for the wrong thing.

Intent blindness

The agent solves what you said, not what you meant. Underspecification compounds across iterations.

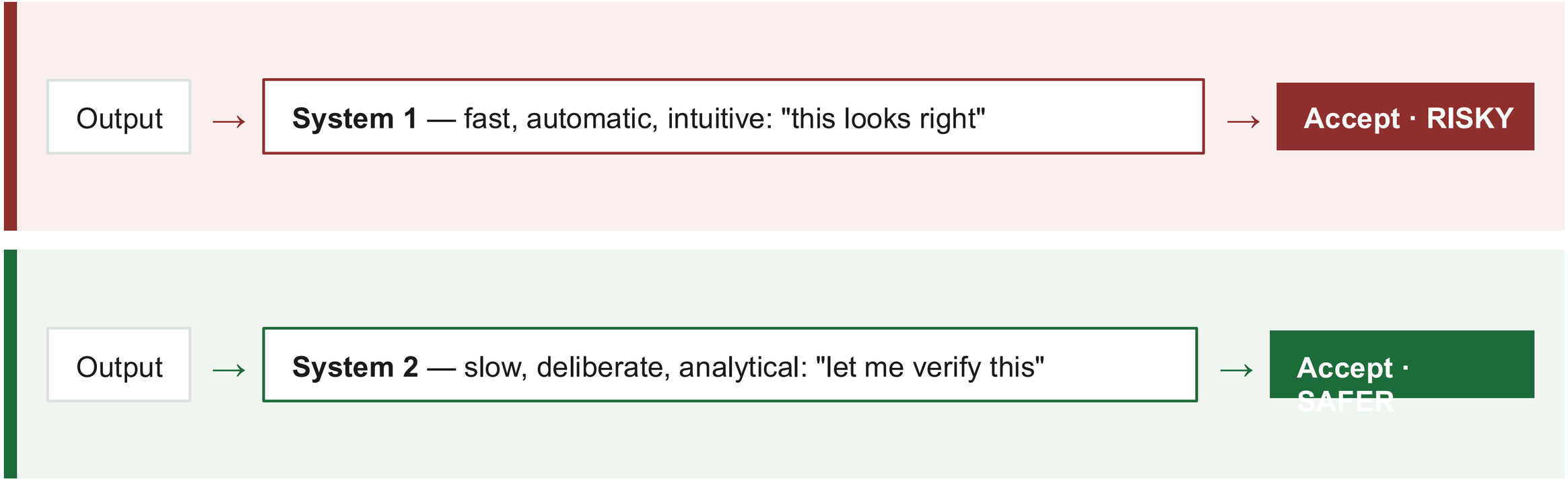
Context window limits

Cannot hold an entire large codebase at once. The agent must choose what to read — and may choose wrong.

Thin training data is the common cause — and a second model has the same blind spots.

Cognitive surrender

"Accepting AI output without engaging System 2 thinking." — Shaw & Nave, 2026



Cognitive surrender: cognitive debt accumulates invisibly. The team feels competent. It may not be.

Cognitive offloading (using a tool for a discrete task) **≠ cognitive surrender** (bypassing your own judgment).

The three debts

Technical debt — lives in the code

Messy code, shortcuts, architectural compromises. Visible. Familiar. Manageable.

Cognitive debt — lives in people

The team's shared understanding of the system erodes. Hard to detect until it is too late.

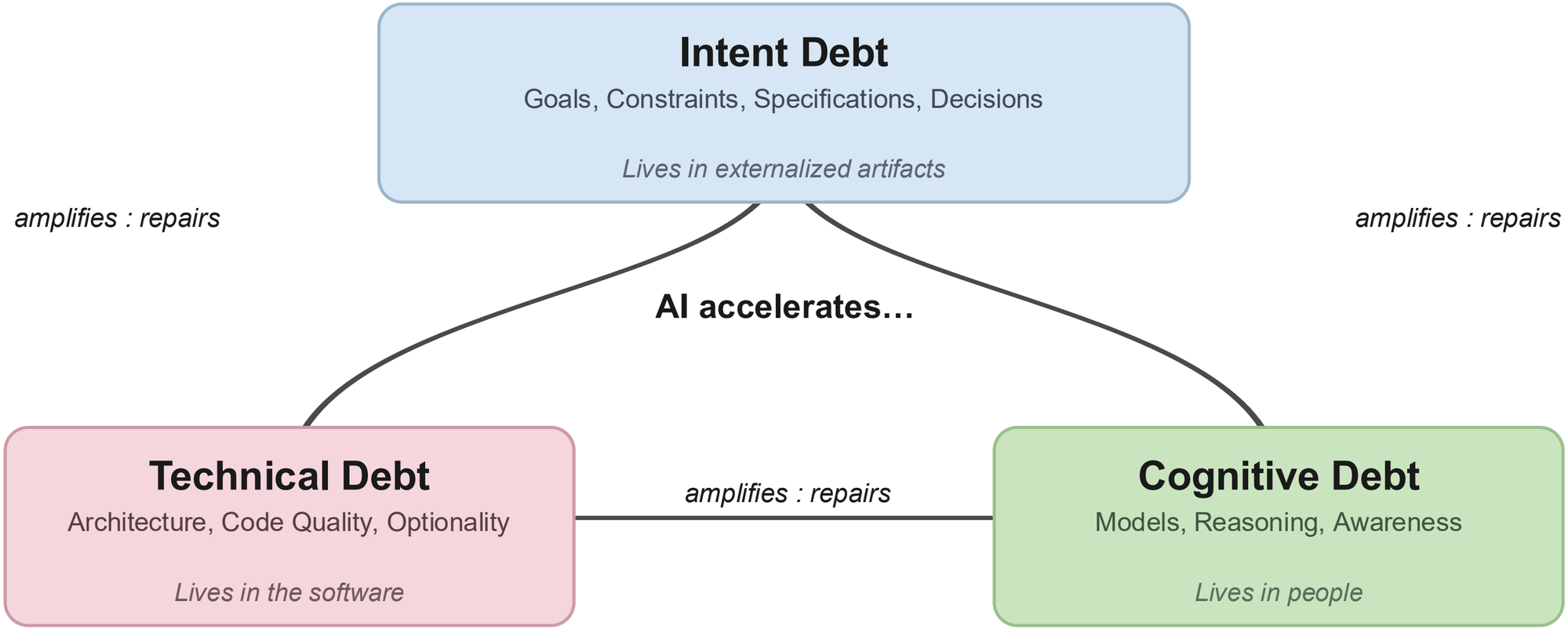
Intent debt — lives in artifacts

Goals, constraints, and specifications never captured. Neither humans nor AI agents can work from what is not written down.

"AI may reduce technical debt while accelerating cognitive and intent debt." The three reinforce each other — the next slide shows how.

The Triple Debt Model

Triple Debt Model for Reasoning about Software Health in the age of AI



Storey, M-A. "From Technical Debt to Cognitive and Intent Debt." arXiv:2603.22106, 2026 — Figure 1.

The code may work. The team may not understand it. And no one wrote down what it was supposed to do.

This is the risk of AI-assisted development.

Practical patterns — how to actually use an agent

Explore → Plan → Implement → Commit

Explore

Read-only. Agent reads the codebase, no changes. Builds understanding. You observe.

→

Plan

Agent proposes a concrete implementation plan. Open questions go to you.

YOU approve and/or adapt

→

Implement

Agent codes against the approved plan. Runs tests as it goes.

→

Commit

Agent stages only touched files. Writes a descriptive message. Opens a PR.

Source: Claude Code best-practice workflow (Anthropic, 2026)

The workflow on a real bug

Task: "Excalidraw's export-to-PNG silently fails for canvases larger than 4000×4000 px."

	WHAT YOU TYPE	WHAT THE AGENT DOES
Explore	`Read the export-to-PNG module and its tests. Don't write any code yet.`	Reads the export module and its tests. No changes.
Plan	`Think about why exports fail above 4000×4000. Show me a plan first, with any open questions — don't change files yet.`	Reproduce with an oversized canvas → locate the failing render call → add a guard and a test. **Open question:** "Cap the size, or tile it?" You answer, then approve.
Implement	`Implement the approved plan. Run the export tests after each change.`	Writes the guard and the test. Runs the export tests as it goes.
Commit	`Commit only the files you touched, referencing the issue, then open a PR.`	Stages only touched files. Message references the issue. Opens a PR.

Plan before execution

```
# one form: a plan file · gitea/modules/indexer/code

## Open questions
1. Should search results be cached, or recomputed each page?
2. Is there an existing pagination helper elsewhere in the
codebase?

## Answers (yours)
1. Recompute for now.
2. Yes — reuse the helper in pagination.go.

## Plan
1. Add a regression test reproducing missing page-2 results.
2. Trace query builder to find where page is dropped.
3. Fix using the shared pagination helper in pagination.go.
4. Re-run the full indexer test suite.
```

The pattern

Explore, form a plan, surface it for review — before before touching any file.

The plan can live anywhere

A file (SPEC.md, planning.md), an outline in the chat, chat, or both.

Plan mode

Claude Code: **Shift+Tab** makes the agent read-read-only — explore and propose only. Approve Approve to execute.

Vague vs. scoped

✗ Vague

"Fix the search bug."

✓ Scoped

"In modules/indexer/code, search results do not paginate past page 1 for repos with 10,000+ files. **Reproduce** with the demo dataset, **find where the page parameter is dropped** , **fix it**, and **add a test** covering page 2 of a large result set."

The biggest cause of bad agent output is a vague goal — ambiguity compounds.

The verification loop

Two verification types — both required.

Automated

inner loop · fast · agent-driven

agent → code change → tests · lint · build → pass? continue : fix ↺

Human — you

outer loop · slow · required

agent output → human review → approved? commit : iterate ↺

Good agents run tests, lint, and build after each change, and react to failures before moving on.

Human review is required for: security-critical code, authentication and access control, anything hard to reverse.

Task complete ≠ correct ≠ understood. All three matter.

Git as a checkpoint strategy

- Commit at logical checkpoints, not when the agent says it is done.
- A bad change is a one-line revert, not an archaeology project.
- Checkpoint before anything hard to undo: schema, auth, dependencies.

```
✓ meaningful checkpoints
a1c9f2 add failing test for page-2 results
7d3e81 reuse shared pagination helper
b52a04 guard oversized export, add test
```

```
✗ one massive commit
3f0b7c fix stuff (142 files changed)
```

Project instructions file

```
# CLAUDE.md — project notes for AI agents

- Run tests with go test ./... before opening a
PR.
- Follow existing error-handling patterns in
modules/; no new error types without discussion.

- If execution diverges from the approved plan,
stop and re-enter Plan Mode.
- Never modify .github/workflows/ without flagging
it in the PR description.
- Anything touching auth or secrets: flag and wait
for human review.
```

The agent reads this at session start — every session.

What it is

Standing context, so you never re-explain it.

Your team will create and maintain one starting Sprint 0.

Spec-driven development

#41277 Open excalidraw/excalidraw

Export fails for canvases > 4000×4000 px

Steps to reproduce: open canvas, draw, export-to-PNG

Expected: export succeeds, or shows a clear size-limit error

Acceptance criteria:

1. Exports up to 8000×8000 px succeed.
2. Larger canvases show an explicit message instead of failing silently.
3. A regression test covers both cases.

→ This is a spec. Feed it to the agent.

Write the spec first. Code is a generated, verifiable artifact of the spec.

A well-written issue is already a small spec.

The agent’s job is to satisfy the acceptance criteria, not to guess what "fails" means.

> **Connection:** this reduces intent debt — goals are now externalized.