The AI Systems Guide

Where each rule should live when you work with Claude Code and Codex. The companion to "What AI Systems Do I Use for Work?" on Writings by Lala.

> A rule that lives only in the chat gets summarized away. A rule that lives in a file comes back every session, and a rule that lives in a setting or a script holds even when the model drifts.

Anthropic's permissions documentation says it directly: "Permission rules are enforced by Claude Code, not by the model. Instructions in your prompt or `CLAUDE.md` shape what Claude tries to do, but they don't change what Claude Code allows."

Where each rule lives

Principle	Where it lives	Failure it prevents	Page
The root file routes and does not teach	`AGENTS.md`, imported by `CLAUDE.md`	Instructions lost in long sessions	3
Permissions are settings, not requests	Allow and deny rules, plus hooks	Constant prompts, risky commands	4
Research happens outside the main chat	Subagents that return a summary	Token burn	5
Edits stay inside their scope	Write boundaries, worktrees	Broken unrelated files	6
"Done" needs evidence	A done checklist and a Stop hook	False completion	7
Corrections carry a reason	Memory files with Why and How to apply	The same mistake next week	8
Facts, links and flags are checked	Source rules, live link checks, `--help`	Invented details	9
Workflows are folders	Numbered stages, one `CONTEXT.md` each	Runs that differ every time	10
Two tools share one rulebook	One `AGENTS.md`, shared skills	Claude and Codex drifting apart	11
Routing is a table	A skill index plus a prompt hook	The wrong workflow	12
Banned words live in one file	A list every draft is scanned against	Writing that reads as AI	13
Access is separate from authority	Deny rules and approval gates	Publishing nobody approved	14
Look before asking	A written rule and a one-line handoff	Questions the agent could answer	15

Pages 16 to 19: starter files and checklists. Page 20: sources, checked live on September 19, 2026, with commands checked against Codex CLI 0.155.0. Tools change, so confirm against your version.

How do I set this up in my first week?

Do one step a day. Each is meant to fit in one sitting. Skip a day if the problem it solves is one you do not have.

> **Day 1. Shrink the rule file.** Run `/context`. Count the lines in `CLAUDE.md`. Keep triggers, pointers and hard boundaries, move procedures into files next to the work, and put the shared rules in `AGENTS.md` with `@AGENTS.md` as the first line of `CLAUDE.md`. Starter on page 16.

> **Day 2. Write permission rules.** Run `/permissions`. Add allow rules for commands you approve every day and a deny rule for `git push`. Starter on page 17.

> **Day 3. Keep progress on disk.** Create a progress file and ask the agent to update it as it works. Use `/clear` between unrelated tasks and `/compact` with a focus when you continue.

> **Day 4. Define done.** Add a "Done means" section to `AGENTS.md` with the check for each kind of task, and ask for the command output in every reply. Checklist on page 19.

> **Day 5. Save one correction properly.** The next time you correct the agent, save it with a Why and a How to apply, and ask for a readback. Template on page 17.

> **Day 6. Turn one workflow into folders.** Pick the task you repeat most. Give each stage a numbered folder, a `CONTEXT.md` and an `output/` folder. Template on page 18.

> **Day 7. Review the week.** List every correction you typed more than once. Promote each into `AGENTS.md`. Anything that recurs after that is a candidate for a hook or a check script.

What to leave for later

- **Hooks.** Keep a rule as text until you have seen it fail. Then add the hook that checks it.
- **Bypass modes.** Skip them outside an isolated container or VM. Anthropic lists `bypassPermissions` for isolated containers and VMs only, and Codex's docs mark running with no sandbox and no approvals as "not recommended."
- **A big skill library.** Every listed skill costs context on every turn. Add a skill when a task repeats, and remove the ones you never use.

Why is Claude ignoring my instructions or hallucinating after 10 to 15 messages?

WHERE THE FIX LIVES

A short root file that routes, procedures in files next to the work, progress in output files, and a hook that restates the one rule that matters.

WHAT THE DOCS SAY

Near the context limit, Claude Code summarizes the conversation. Anthropic's memory docs: the project-root `CLAUDE.md` is re-read from disk afterward, and "If an instruction disappeared after compaction, it was given only in conversation," or sits in a file that has not reloaded yet. They also say: "target under 200 lines per CLAUDE.md file. Longer files consume more context and reduce adherence."

1. Run `/context` to see what is filling the window.
2. If `CLAUDE.md` runs past 200 lines, keep triggers, pointers and hard boundaries. Move each procedure into a file next to the work it governs and leave a one-line pointer.
3. Switching to unrelated work: `/clear`. Continuing the same work: `/compact` with a focus.
4. Write progress to a file as you go, so the next session reads state from disk.
5. In Codex, `/compact` summarizes the visible chat and `/status` shows how much context remains.
6. Later, add a `UserPromptSubmit` hook that injects your single most important rule. Keep it short, because it rides along on every message.

```
/context
/compact Keep the list of changed files and the failing tests
/clear
```

```
# CLAUDE.md
@AGENTS.md

# Claude-only notes
- Progress lives in progress.md. Read it first when resuming.
```

CHECK THAT IT WORKED

Start a fresh session and ask what the project rules are. The answer should match `AGENTS.md` without you pasting anything.

How do I keep Claude Code from asking permission for every file read or git status?

WHERE THE FIX LIVES

Permission rules in settings files, enforced by the tool, with deny rules for anything the agent must never run and ask rules for anything you want to approve.

WHAT THE DOCS SAY

File reads inside the working directory need no approval, and read-only commands such as `ls`, `cat`, `grep`, `find` and read-only `git` run without a prompt. In Manual mode, Claude Code still asks when it cannot fully parse a command, or for `git` with an unquoted glob. Rules are evaluated deny, then ask, then allow.

1. Run `/permissions` to see every rule and the file it comes from.
2. For a command you trust, choose "Yes, and don't ask again." The rule is saved to `.claude/settings.local.json`.
3. Put team-wide rules in `.claude/settings.json`.
4. Put the `*` after the subcommand. `Bash(git log *)` allows only `git log`. `Bash(git *)` allows every git command.
5. Press Shift+Tab to cycle modes. `acceptEdits` auto-approves reads, file edits and common filesystem commands. `plan` explores without editing source files.
6. In Codex, `codex --sandbox workspace-write --ask-for-approval on-request` is the Auto preset, and `/permissions` changes it mid-session.

```
{
  "permissions": {
    "allow": ["Bash(npm run *)", "Bash(git commit *)"],
    "deny": ["Bash(git push *)", "Read(./.env)"]
  }
}
```

A `Read` deny covers Claude's file tools and commands such as `cat`, not a script that opens the file itself. The docs point to the sandbox for that.

For commands Codex runs outside its sandbox, a rules file with `prefix_rule` can allow, prompt or forbid them. The starter is on page 17.

CHECK THAT IT WORKED

In a scratch repository, ask the agent to run `git push --dry-run`. The deny rule should refuse it. If it runs anyway, the dry run sends nothing, and the rule needs fixing.

How do I stop burning through my entire token allowance in an afternoon?

WHERE THE FIX LIVES

Research in subagents that return a summary, instructions in skills that load only when used, and a routing table that sends each request to one file.

WHAT THE DOCS SAY

Anthropic's costs page: "Token costs scale with context size" and "Stale context wastes tokens on every subsequent message." Each subagent runs in its own context window. A skill's full content loads only when invoked, but every listed skill adds to context on each turn. CLI tools are more context-efficient than MCP servers because they add no per-tool listing.

1. Run `/usage` at the start and end of a work block. In Codex, `/status` shows token usage.
2. Use `/clear` between unrelated tasks. Run `/rename` first if you want to return with `/resume`.
3. Send research and wide searches to a subagent with a complete brief (below) and a length cap.
4. Move rarely used instructions from `CLAUDE.md` into a skill. In recent versions, `/skill-doctor` shows what each skill costs, so you can turn off the ones you never use.
5. Prefer a command-line tool such as `gh` or `gcloud` over an MCP server that does the same job.
6. Watch for loops. If the agent has searched five times in two minutes without progress, stop it and ask what it is looking for.

```
Subagent brief
Question:   Which files call the old payments client?
Look in:    src/, scripts/
Done when:  every call site is listed with its file path and line
Reply:      under 200 words, list format, no file contents
```

CHECK THAT IT WORKED

Run `/context` after a research task sent to a subagent. The main conversation should hold only its summary, not the raw results.

How can I stop Claude from breaking unrelated files during refactors?

WHERE THE FIX LIVES

Written write boundaries in `AGENTS.md`, a worktree for the refactor, deny rules for protected paths, and a diff review before commit.

WHAT THE DOCS SAY

`claude --worktree <name>` or `-w` creates an isolated worktree under `.claude/worktrees/<name>/` on a new branch. `/rewind`, or Esc twice on an empty prompt, rolls back Claude's edits, but changes made by Bash commands are not tracked. In Codex's default `workspace-write` sandbox, `.git` is read-only.

1. Commit or stash first, so `git diff` shows only the agent's changes.
2. Start in a worktree: `claude -w refactor-auth`, or `codex --worktree`.
3. Name the scope in the request and ask the agent to stop and ask before touching anything outside it.
4. Add the write boundaries below to `AGENTS.md`.
5. Deny edits to paths that must stay untouched, such as `Edit(/db/migrations/**)`.
6. Before committing, run `git diff --name-only` and question every file you did not expect.
7. Keep throwaway experiments in a temporary folder from `mktemp -d`, outside the project.

```
## Write boundaries
- Change only the files or folders named in the request.
- Preserve unrelated changes. Patch shared files narrowly.
- Never revert a file you did not change.
- Never assume another session's changes are disposable.
- Never edit node_modules/, venv/ or .git/.
```

CHECK THAT IT WORKED

`git diff --name-only` lists only files inside the scope you named.

How do I stop Claude or Codex from saying "done" when nothing was checked?

WHERE THE FIX LIVES

A "Done means" section in `AGENTS.md`, evidence in every reply, a goal written like a contract, and later a Stop hook.

WHAT THE DOCS SAY

OpenAI's Codex guide lists "What done means and how to verify work" as part of a good `AGENTS.md`. Claude Code hooks run at session, turn and tool-call events, including `Stop`.

1. Add a "Done means" section with the check for each kind of task.
2. Allow the words done, fixed and verified only after the matching check ran.
3. Ask for the evidence in the reply: the command and its output, or the path of the file opened.
4. For multi-step work, write the goal with the five parts below before starting.
5. Keep the rule as text until you see it fail. Then add a `Stop` hook that checks for it.
6. Ask for one final line in every reply that says what the agent needs from you.

```
## Done means
- Code: npm test passes; quote the last lines of output.
- Page: open it and confirm the expected heading is visible.
- Data: read back the exact rows or range you wrote.

## Goal
Output:        docs/release-notes.md
Verify with:   npm run lint:docs
May change:    docs/ only
Stop when:     the verifier passes and the file lists every merged PR
Pause if:      a PR description is missing or unclear
```

CHECK THAT IT WORKED

Every "done" in the reply sits next to a command, an output or a file path you can check yourself in under a minute.

How do I make Claude remember a correction in the next session?

WHERE THE FIX LIVES

A memory file per correction, with the rule, a Why and a How to apply, plus a short index that loads every session.

WHAT THE DOCS SAY

Claude Code's auto memory keeps a `MEMORY.md` index plus topic files. The first 200 lines of `MEMORY.md`, or the first 25KB, load at the start of every conversation. OpenAI's Codex guide: "add new rules only after you notice repeated mistakes."

1. When you correct the agent, ask it to save the correction with a Why and a How to apply.
2. Ask it to read the saved file back to you. Count it as saved only after the readback.
3. Keep `MEMORY.md` to one line per memory, with the detail in the topic file.
4. The second time you make the same correction, move it into `AGENTS.md` or `CLAUDE.md`.
5. If it still recurs, write it as a warning first, then a check or hook once the warning has proved itself.

```
---
name: verify-before-done
description: Check the output against the original request before saying done
metadata:
  type: feedback
---

Before reporting a task complete, restate the request and match
each part to evidence.

**Why:** Work passed its own checks while drifting from what was asked.

**How to apply:** Name the output and the check that proves it in
the final reply.
```

CHECK THAT IT WORKED

In a new session, repeat the situation that caused the correction. The agent should apply it without being told.

How do I stop an AI agent from inventing facts, links or commands?

WHERE THE FIX LIVES

Source rules in `AGENTS.md`: label what is inferred, open every link before listing it, and run a tool's help before quoting a flag.

A REAL CASE

While preparing the article, the plan included Codex's `--full-auto` flag. Neither `codex --help` nor `codex exec --help` for Codex 0.155.0 listed it, and OpenAI's current docs describe `codex exec --full-auto` as "a deprecated compatibility path." The help check caught it before it was published.

1. Add the three source rules below to `AGENTS.md`.
2. Ask for the source beside each claim: a file path, a URL or the command output.
3. Separate what was observed from what was inferred and what is still unknown.
4. Before anything is published, run a separate pass that opens each link and runs each command's help, ideally in a subagent with fresh context.
5. Treat a summary of a page as a lead, then open the page itself. A summarizing fetch can paraphrase a quote or miss a line.

```
## Source rules
- Open every link live before listing it.
- Run the tool's --help (and the subcommand's) before quoting a flag.
  Use only flags that appear there.
- Label every claim as observed, instructed, inferred or unknown.
- Quote documentation only from the page itself, not from a summary.
```

```
codex --help
codex execpolicy check --help
curl -sL -o /dev/null -w '%{http_code}\n' https://example.com/page
```

CHECK THAT IT WORKED

Pick three claims from the agent's last answer at random. Each should come with a source you can open.

How do I get an AI workflow to run the same way every time?

WHERE THE FIX LIVES

Numbered stage folders, a `CONTEXT.md` in each, output folders that carry state, and a template copied for every run.

WHERE THE PATTERN COMES FROM

ICM, the Interpretable Context Methodology, by Jake Van Clief and David McDermott (arXiv 2603.16021, March 2026): "Numbered folders represent stages. Plain markdown files carry the prompts and context that tell a single AI agent what role to play at each step."

1. Pick one workflow you repeat every week.
2. Create numbered folders for its stages, each with a `CONTEXT.md` and an `output/` folder.
3. In each `CONTEXT.md`, write one job, what to load, what to skip, what to write and the exit question.
4. Keep a template copy and start every run by copying it.
5. Let the files carry state. An empty `output/` means the stage has not run.
6. Later, add a script that checks the previous stage's output before the next stage starts.

```
my-workflow/
  CLAUDE.md                  map and routes only
  _templates/run/            the copy each new run starts from
  runs/2026-09-18/
    01_research/CONTEXT.md   output/
    02_outline/CONTEXT.md    output/
    03_draft/CONTEXT.md      output/
```

The full `CONTEXT.md` template is on page 18.

CHECK THAT IT WORKED

Run the workflow twice on different inputs. Both runs should produce the same set of output files, in the same order.

How do I run Claude Code and Codex on one project without two sets of rules?

WHERE THE FIX LIVES

One `AGENTS.md` at the project root, a `CLAUDE.md` that imports it, and one copy of each skill linked into each tool.

WHAT THE DOCS SAY

Anthropic: "Claude Code reads `CLAUDE.md`, not `AGENTS.md`," so create a `CLAUDE.md` that imports it. Codex reads one global file in `~/.codex`, then walks from the project root to the current folder; closer files override earlier ones, empty files are skipped, and it stops at 32 KiB by default.

1. Put the shared rules in `AGENTS.md` at the project root.
2. Create `CLAUDE.md` with `@AGENTS.md` as its first line. Add Claude-only notes below it.
3. Keep the shared file short enough for both tools: under 32 KiB for Codex, and under the 200-line target for Claude.
4. Keep one copy of each skill and link it in. Claude Code personal skills live in `~/.claude/skills/<skill-name>/`, and the entry can be a symlink. OpenAI's current docs list `~/.agents/skills` for personal Codex skills. Check the path for your version, and confirm in a new Codex session that a linked skill loads; if it does not, copy the folder instead.
5. Give each tool, or each terminal, its own progress file, so parallel sessions do not overwrite each other.

```
mkdir -p ~/ai-skills/release-notes ~/.claude/skills ~/.agents/skills
# write ~/ai-skills/release-notes/SKILL.md once, then link it:
ln -s ~/ai-skills/release-notes ~/.claude/skills/release-notes
ln -s ~/ai-skills/release-notes ~/.agents/skills/release-notes
```

CHECK THAT IT WORKED

Change one rule in `AGENTS.md`, start a new session in each tool, and ask what the rule says. Both should give the new answer.

How does Claude know which skill or workflow a request needs?

WHERE THE FIX LIVES

Skill descriptions written as triggers, a routing table the agent consults before acting, and suggest-then-confirm for new request types.

WHAT THE DOCS SAY

In Claude Code, skill descriptions load into context and a skill's full content loads only when it is used. Codex starts with each skill's name and description and loads the full `SKILL.md` when it decides to use it.

1. Rewrite each skill description to include the phrases you actually say when you need it.
2. Keep a routing table with five columns in `AGENTS.md` or in a file it points to.
3. Add one rule: consult the table before starting any task.
4. For new request types, ask the agent to propose its plan and wait for your yes. Keep an override phrase such as "just answer."
5. If you keep typing the same workflow name, a `UserPromptSubmit` hook can match it and inject "open this file first."

```
| Trigger            | Owner   | Open first                  | Status | Done when          |
|--------------------|---------|-----------------------------|--------|--------------------|
| release notes      | docs    | docs/workflows/release.md   | active | lint:docs passes   |
| new blog post      | writing | blog/01_intake/CONTEXT.md   | active | final check passes |
| weekly report      | reports | reports/weekly/CONTEXT.md   | active | links all open     |
```

```
---
name: release-notes
description: Write release notes from merged PRs. Use when I say
  "release notes", "changelog", or "what shipped this week".
---
```

CHECK THAT IT WORKED

Type a request in your own words. The agent should name the right file to open before it starts.

How do I make AI-assisted writing stop sounding like AI?

WHERE THE FIX LIVES

One banned-list file that every prompt and voice profile points to, a scan on every draft, and a checklist for shapes that need judgment.

HOW MINE IS BUILT

Two punctuation marks, 48 words and phrases, 5 patterns a human must judge and 7 structural shapes. Em dashes and exclamation points fail automatically. The list came from tells I noticed in AI drafts myself.

1. Start a list from your own edits. Any word or phrase you delete twice goes on it.
2. Keep it in one file that every prompt and profile points to.
3. Store punctuation by its Unicode codepoint, so the file does not fail its own check.
4. Scan each draft before reading it closely. The command below matches inside longer words too, so expect a few false hits. Keep blank lines out of the list, or every line will match. This scan covers words; punctuation needs its own check.
5. Keep the structural shapes as a review checklist. They need judgment, and a word count will not catch them.
6. For a target voice, write down a writer's reasoning moves, not their vocabulary.

```
grep -n -i -F -f banned-words.txt draft.md
```

Structural shapes to review by hand

- A three-item list used as a default rhythm instead of because the subject has three parts.
- A paragraph that resolves into a matched pair.
- A tidy one-line close on every section.
- Repeated "no" and "not" used as decoration.
- Every paragraph the same length.
- A generic noun where the source named something specific.
- A sentence that announces what comes next instead of saying it.

CHECK THAT IT WORKED

The word scan returns nothing, and a read-aloud turns up none of the seven shapes more than once.

Where should a human approve AI agent work, and where can the agent run alone?

WHERE THE FIX LIVES

Deny or ask rules on anything public or hard to reverse, automatic movement for reversible local work, and approval of exact output.

WHAT THE DOCS AND MY RULES SAY

Anthropic's permission-mode docs: "Deny rules block in every mode, including `bypassPermissions`."
My own rule: "Publishing access is not publishing authority."

1. List the actions in your work that are public or hard to reverse: publishing, sending, deleting, paying, pushing.
2. Put those behind deny or ask rules.
3. Let reversible local work run without prompts, such as reading files or running tests inside a worktree.
4. Approve exact output. Record the fingerprint with your approval, and approve again if it changes.
5. A "yes" applies only to the proposal right before it. Say so in `AGENTS.md`.
6. Write the pause condition into every larger task.

```
# on a Mac: fingerprint the exact draft you approved
shasum -a 256 draft.md

# AGENTS.md
## Approval
- Publishing, sending, deleting, paying and pushing need my explicit yes.
- A yes covers only the proposal immediately before it.
- If the approved file changes by one byte, ask again.
```

```
"deny": ["Bash(git push *)"],
"ask":  ["Bash(npm publish *)"]
```

CHECK THAT IT WORKED

In a scratch repository, ask the agent to run `git push --dry-run`. A deny rule should refuse it in every mode. For an ask rule, confirm the prompt appears in the mode you normally use.

How do I stop Claude from asking me questions it could answer by looking?

WHERE THE FIX LIVES

A written look-first rule, a resume rule that reads the progress file, and one closing line that separates real questions from status.

HOW MINE WORKS

When I say "you have everything you need" or "why are you asking me," the agent must exhaust the safe files, credentials and tools in scope before asking me to take over, and come back with one grounded action instead of a menu. Every reply ends with "What I need from you:" followed by the real request or "Nothing right now."

1. Add a line to `AGENTS.md` naming where to look before asking.
2. Tell the agent to resume from the progress file without asking where you left off.
3. Separate decisions from facts: the agent should ask you only for a decision, never for a fact it can look up.
4. Ask for one action, not a list of options, when the answer is findable.
5. End every reply with one line of what is needed from you, so a real question does not hide inside a status update.

```
## Before asking me
- Look in the project files, progress.md, docs/ and the tool's --help.
- If I say "you have everything you need", exhaust those first and
  return one grounded action, not a menu.

## Resuming
- Read progress.md first. Do not ask where we stopped.

## End every reply with
What I need from you: <the request, or "Nothing right now.">
```

CHECK THAT IT WORKED

Count the questions in the agent's next five replies. Each remaining one should be a decision only you can make.

Starter files: AGENTS.md and CLAUDE.md

Copy these into the root of a project. Replace the example paths and commands with your own. Keep `AGENTS.md` short: it routes and sets boundaries, and the procedures live in the files it points to.

```
# AGENTS.md
# Project rules, shared by Claude Code and Codex.

## Where things live
- Procedures: docs/workflows/<task>.md. Read the one that matches the task.
- Progress: progress.md. Read it first when resuming, update it as you work.
- Routing table: docs/routing.md. Consult it before starting any task.

## Write boundaries
- Change only the files or folders named in the request. Ask before others.
- Preserve unrelated changes. Never revert a file you did not change.
- Never edit node_modules/, venv/ or .git/.

## Done means
- Code: npm test passes; quote the last lines of output.
- Pages: open the page and confirm the expected heading is visible.
- Say done, fixed or verified only after that check ran.

## Source rules
- Open every link before listing it. Run --help before quoting a flag.
- Label anything inferred.

## Approval
- Publishing, sending, deleting, paying and pushing need my explicit yes.
- A yes covers only the proposal immediately before it.

## Before asking me
- Look in the project files, progress.md, docs/ and --help first.

## End every reply with
What I need from you: <the request, or "Nothing right now.">
```

```
# CLAUDE.md
@AGENTS.md

# Claude-only notes
- Send research and wide searches to a subagent. Ask for a summary
  under 200 words.
- When compacting, keep the list of changed files and open questions.
```

Starter files: settings, Codex rules and a memory template

.claude/settings.json

```
{
  "permissions": {
    "allow": ["Bash(npm run *)", "Bash(git commit *)"],
    "ask":   ["Bash(npm publish *)"],
    "deny":  ["Bash(git push *)", "Read(./.env)", "Edit(/db/migrations/**)"]
  }
}
```

Deny is checked first, then ask, then allow. Personal approvals from "Yes, and don't ask again" go to `.claude/settings.local.json`.

~/.codex/rules/default.rules

```
prefix_rule(
    pattern = ["git", "status"],
    decision = "allow",
)
prefix_rule(
    pattern = ["git", "push"],
    decision = "forbidden",
)
```

Test with `codex execpolicy check --pretty --rules ~/.codex/rules/default.rules -- git push`. OpenAI marks rules as experimental.

A memory file for one correction

```
---
name: short-kebab-case-name
description: One line that says when this applies
metadata:
  type: feedback
---

The rule, in one or two sentences.

**Why:** What went wrong, or what the person said.

**How to apply:** When and where this rule should change the work.
```

Starter files: a stage-folder workflow

Copy the template folder for every new run. Each stage reads only what its `CONTEXT.md` names and writes only to its own `output/`. An empty `output/` means the stage has not run.

```
my-workflow/
  CLAUDE.md                  map and routes only
  CONTEXT.md                 the pipeline on one screen
  _templates/run/
    01_research/CONTEXT.md   output/
    02_outline/CONTEXT.md    output/
    03_draft/CONTEXT.md      output/
    04_review/CONTEXT.md     output/
  runs/
    2026-09-18/              a copy of _templates/run/
```

A CONTEXT.md for one stage

```
# 02_outline

## One job
Turn the research notes into an outline the draft can follow.

## What to load
- ../01_research/output/notes.md
- The request in ../../REQUEST.md

## What to skip
- Everything in later stages.
- Old outlines from other runs.

## What to write
- output/outline.md

## Move on when
Every section in the outline traces to a note, and the exit line
at the bottom of output/outline.md reads: Exit result: pass
```

A one-screen pipeline map (root CONTEXT.md)

```
| # | Stage       | One job                        | Move on when                    |
|---|-------------|--------------------------------|---------------------------------|
| 01| research    | Collect sources and notes      | every note has a source         |
| 02| outline     | Turn notes into an outline     | every section traces to a note  |
| 03| draft       | Draft from the outline         | the draft follows the outline   |
| 04| review      | Check facts, links and words   | the checks pass; I approve      |
```

Checklists: done, and a banned-words starter

Before anything is called done

- Restate what was requested, in one or two lines.
- Map each requirement to its evidence: a file, a command result, a readback or a visible page.
- Open, render or look at anything visual.
- Run the relevant check: tests, build, lint or the workflow's own verifier.
- Open every link that will be shown to someone else.
- Read back any data written to a spreadsheet or database.
- Say what was not checked, instead of letting a passing check imply more than it proved.
- End with one line of what is needed from the person, or "Nothing right now."

Once a week

- List every correction you typed more than once. Each one belongs in `AGENTS.md` or a memory file with a Why.
- Check that `CLAUDE.md` is still under 200 lines and `AGENTS.md` still routes instead of teaching.
- In recent versions, run `/skill-doctor` and remove skills you never used.
- Any written rule that failed twice is a candidate for a warning, then a hook.

banned-words.txt, a starter format

One entry per line. Start with your own deletions. The three below are examples only, not a recommended list.

```
leverage
seamless
robust
```

Punctuation by codepoint (banned-punctuation.json)

```
{
  "punctuation": [
    {"name": "em dash", "codepoint": "U+2014",
     "instead": "use a comma, a full stop, or a new sentence"}
  ]
}
```

Storing the codepoint instead of the character keeps the list itself from failing the scan that reads it.

Sources, checked September 19, 2026

Every tool claim in this guide comes from these pages, from the live help of Codex CLI 0.155.0, or from the help of the local command-line tools. Each URL returned HTTP 200 when checked.

Anthropic, Claude Code documentation

- Memory and CLAUDE.md: https://code.claude.com/docs/en/memory
- Permissions: https://code.claude.com/docs/en/permissions
- Permission modes: https://code.claude.com/docs/en/permission-modes
- Managing costs: https://code.claude.com/docs/en/costs
- Hooks: https://code.claude.com/docs/en/hooks
- Subagents: https://code.claude.com/docs/en/sub-agents
- Skills: https://code.claude.com/docs/en/skills
- Worktrees: https://code.claude.com/docs/en/worktrees
- Checkpointing: https://code.claude.com/docs/en/checkpointing
- Commands: https://code.claude.com/docs/en/commands
- Settings: https://code.claude.com/docs/en/settings

OpenAI, Codex documentation

- AGENTS.md: https://learn.chatgpt.com/docs/agent-configuration/agents-md
- Slash commands: https://learn.chatgpt.com/docs/cli/slash-commands
- Skills: https://learn.chatgpt.com/docs/build-skills
- Rules: https://learn.chatgpt.com/docs/agent-configuration/rules
- Best practices: https://learn.chatgpt.com/guides/best-practices
- Approvals and security: https://learn.chatgpt.com/docs/agent-approvals-security

The folder pattern

- Jake Van Clief and David McDermott, "Interpretable Context Methodology: Folder Structure as Agentic Architecture," arXiv 2603.16021, March 2026: https://arxiv.org/abs/2603.16021

The article

- "What AI Systems Do I Use for Work?" on Writings by Lala, which explains where each rule lives in my own setup and why.

If you change only one thing this week, take the correction you have typed most often and give it a home outside the chat. Then check whether it still holds a week later.