

The Only Cursor Guide You'll Ever Need

Every capability. Every configuration file. From zero to a fully autonomous development environment · in one document.

Rules. Skills. Subagents. MCP servers. Hooks. Plugins. Browser automation. AGENTS.md. Four modes. Best practices. Nothing left out.

Why This Guide Exists

There are plenty of Cursor guides online. Most cover one or two features · rules, MCP setup, or agent workflows. None of them cover everything in one place.

This guide does. It walks through **every configuration file Cursor supports**, explains when and why you'd use each one, provides real examples you can adapt, and gives you a step-by-step path from "I just installed Cursor" to "my entire team shares a project-aware AI environment that auto-formats code, blocks dangerous commands, fetches live documentation"

What makes this different from every other Cursor guide:

| What others cover | What this guide adds | |---|---| | Rules (.mdc files) | Rules + **Skills** + **Subagents** + **Hooks** + **Plugins** + **AGENTS.md** · the complete system | MCP setup (1-2 servers) | **20+ free MCP servers** categorized by use case, with install commands | | "Use Agent mode" | All **four modes** (Agent, Ask, Plan, Debug) with when-to-use guidance | | Generic tips | **Real .cursor/ folder anatomy** · every file, what it does, why it's there | | No mention of browser testing | Full **browser automation** workflow via Playwright MCP | | No mention of hooks | **Lifecycle hooks** with script examples for auto-formatting and safety guardrails | | No mention of plugins | **Plugin ecosystem** · marketplace, Continual Learning, Team Kit, custom plugins |

If you follow this guide end to end, you'll have a Cursor setup that most developers don't even know is possible.

Based on Cursor Docs, Cursor's best practices blog, and hands-on implementation across a multi-pillar project (Python SDK + web app + research notebooks + documentation site).

Part 0: Getting Started (For Beginners)

If you've never used Cursor before, start here. If you're already comfortable, skip to The Core Idea.

What Is Cursor?

Cursor is a code editor (built on VS Code) with an AI agent built in. The agent can:

- Read and edit files across your project
- Run terminal commands (tests, builds, git)
- Search your codebase by meaning, not just text
- Control a browser to test web applications
- Connect to external tools (GitHub, docs, databases) via MCP

You interact with it by typing prompts in the chat sidebar (open with `Cmd+I` on Mac, `Ctrl+I` on Windows).

First Steps: Open a Project

```
# Clone any repo and open it in Cursor
git clone https://github.com/your-org/your-project.git
cd your-project
cursor .
```

That last command opens the current folder in Cursor. If `cursor` isn't in your `PATH`, open Cursor manually and use `File · Open Folder`. You can also open any existing folder · Cursor works with any codebase, any language.

Wait for indexing. Cursor indexes your codebase on first open. Don't start chatting until the indexing indicator (bottom-right) finishes. This powers the semantic search that makes the agent useful.

Your First Prompt

Open Agent chat (`Cmd+I`) and type:

```
Explain this codebase. Point me to the main entry points, key
modules,
and anything I should read before making changes.
```

The agent searches your repo, reads files, and gives you a summary. This is the fastest way to get oriented.

The `.cursor/` Folder

This is where all Cursor project configuration lives. When you clone a repo that has a `.cursor/` folder, you inherit the team's AI configuration · rules, MCPs, hooks, everything.

```
your-project/
... .cursor/
  ... rules/           · Rules: persistent AI guidance
  ... skills/          · Skills: repeatable workflows
  ... agents/          · Subagents: specialist workers
  ... mcp.json         · MCP servers: external tools
  ... hooks.json       · Hooks: lifecycle automation
  ... settings.json    · Plugin configuration (see Part 9)
  ... AGENTS.md        · Simple markdown instructions (alternative to
rules)
  ... ..
```

If the repo doesn't have a `.cursor/` folder, you create one. That's what this guide teaches.

The Four Modes

Cursor has four distinct modes. Cycle between them with `Shift+Tab` in the agent input, or use the mode picker dropdown.

| Mode | What It Does | When to Use | |-----|-----|-----| | **Agent** | Reads, edits files, runs commands, uses tools | Default mode · building features, fixing bugs, running tests | | **Ask** | Read-only · explores code, answers questions, **never edits anything** | Understanding code you didn't write, learning how a module works, tracing a flow | | **Plan** | Researches codebase, asks questions, creates a reviewable plan · **no code until you approve** | Multi-file changes, architectural decisions, unclear requirements | | **Debug** | Instruments code with logging, collects runtime evidence, then makes targeted fixes | Tricky bugs you can reproduce but can't figure out · regressions, race conditions, memory leaks |

Ask Mode is the one most people overlook. Use it when you want to *understand* without *changing*:

- "How does the authentication flow work in `app/auth/`?"
- "Where is the database connection configured?"
- "What's the relationship between `BaseModel` and `UserService`?"
- "Explain what `process_request()` does in the `AuthService` class"

The agent searches your codebase, reads files, and explains · but touches nothing. This is the safest way to explore an

unfamiliar codebase, and it's faster than reading files yourself because it follows the call chain for you.

Plan Mode is underused and extremely valuable. For anything touching multiple files, toggle Plan Mode first. The agent researches your codebase, asks clarifying questions, and creates a reviewable Markdown plan *before writing any code*. You can edit the plan, save it to `.cursor/plans/`, and share it with your team.

Key Shortcuts

Action	Mac	Windows	----- -----	Open Agent	Cmd+I	Ctrl+I	Toggle Plan Mode
Shift+Tab in input	Shift+Tab in input	Attach context	Type @	Type @	Queue a message while agent works	Enter	Enter
Send immediately (interrupt)	Cmd+Enter	Ctrl+Enter	Change model	Cmd+ /	Ctrl+ /		

The Core Idea

Cursor's agent starts every session with amnesia. It doesn't know your conventions, your file structure, your preferred libraries, or your testing strategy. Every session, you re-explain the same things.

The fix is a set of configuration files · all living in `cursor/` · that give the agent persistent memory, external tools, and automated guardrails. Here's what each one does:

File/Directory	What It Does	Analogy	----- ----- -----	<code>.cursor/rules/*.mdc</code>	Injects project conventions into every relevant chat	Like <code>.editorconfig</code> but for AI behavior
<code>.cursor/skills/*.SKILL.md</code>	Teaches the agent repeatable workflows	Like shell aliases for complex procedures	<code>.cursor/agents/*.md</code>	Defines specialist sub-assistants with their own context	Like microservices · each one has a single job	<code>.cursor/mcp.json</code>
Connects the agent to external tools and live documentation	Like <code>package.json</code> for AI tool dependencies	<code>.cursor/hooks.json</code>	Runs scripts before/after agent actions	Like git hooks, but for the AI loop	<code>AGENTS.md</code>	Plain-text project instructions (simpler alternative to rules)
Like a README the AI actually reads						

All of these are version-controlled. Commit `.cursor/` to git, and every contributor gets the same AI behavior.

What This Guide Covers

Most real projects aren't monoliths. They have distinct areas · a backend, a frontend, a docs site, a testing harness, a CLI tool, research notebooks. Each area has different conventions, different libraries, and different things the AI needs to know.

Example · a multi-pillar project:

Workstream	Path	Nature of Work	----- -----	SDK / Library	<code>src/</code>	Python/TypeScript library. Has its own patterns, class hierarchies, test strategy.
Web App	<code>app/</code>	Backend (FastAPI/Express) + Frontend (React/Vue). Auth, API routes, components.	Research	<code>docs/</code> , <code>notebooks/</code>	Jupyter notebooks, experiments, benchmarking, external research.	Documentation
<code>website/</code>	DocuSaurus/MkDocs/Astro site. Tutorials, API docs, blog posts.					

The setup below teaches Cursor to behave **differently depending on which part of the codebase you're in**. When you edit a React component, it knows your frontend conventions. When you edit a Python template, it knows your SDK patterns. You configure this once, commit it, and every contributor gets the same AI behavior.

Part 1: Rules · What the AI Should Always Know

Docs: cursor.com/docs/rules

A rule is a markdown file with YAML frontmatter. When you open a file matching a rule's glob pattern, that rule is silently injected into the agent's context. You never have to explain it again.

How to Create a Rule

Two options:

- **From chat:** Type `/create-rule` and describe what you want in natural language
- **Manually:** Create a `.mdc` file in `.cursor/rules/`

The Four Rule Types

Always Apply	· Every chat session, no matter what file is open
Apply Intelligently	· Agent reads the description and decides
Glob-Scoped	· Only when files matching the pattern are in context
Manual	· Only when you type <code>@rule-name</code> in chat

Rules Tailored to Each Workstream

Here's what a complete set of workstream rules looks like. Adapt these to your project.

SDK / library rule · activates when editing library source:

```
---
description: SDK coding conventions
globs: ["src/**/*.py"]
alwaysApply: false
---
- Every public class must have a docstring explaining purpose and
usage
- Use Pydantic v2 for parameter validation
- Depth/verbosity parameters should produce meaningfully different
output per level
- See src/core/base_pattern.py as the reference implementation
- Constraints must be specific ("include severity rating") not
vague ("be thorough")
```

Backend rule · activates when editing API code:

```
---
description: FastAPI backend conventions
globs: ["app/api/**/*.py", "app/services/**/*.py"]
alwaysApply: false
---
- Services in app/services/, routers in app/api/ · keep routers thin
- All routes use Depends() for auth injection
- Settings via pydantic-settings BaseSettings, loaded through
get_settings()
- DB models in app/db/models.py using SQLAlchemy
- All endpoints must be async
```

Frontend rule · activates when editing React components:

```
---
```

```

description: React frontend conventions
globs: ["app/web/src/**/*.jsx,tsx,css"]
alwaysApply: false
---
- Functional components only, no class components
- Auth via useAuth() hook from context/AuthContext
- Routing: react-router-dom, protected routes wrap with
  <ProtectedRoute>
- CSS: per-component .css files, CSS variables for theming
  (var(--accent))
- API calls go through a centralized api/client.js

```

Documentation rule · activates when editing docs site content:

```

---
description: Documentation site conventions
globs: ["website/docs/**/*.md", "website/blog/**/*.md"]
alwaysApply: false
---
- Frontmatter required: sidebar_position, title, description (all
  three, always)
- Code examples must use real project imports · never pseudocode
- Links: relative paths (./installation not /docs/installation)
- Mermaid diagrams: use ```mermaid code blocks

```

Research rule · activates when editing notebooks:

```

---
description: Notebook and experiment conventions
globs: ["docs/**/*.ipynb", "notebooks/**/*.ipynb",
  "research/**/*.ipynb"]
alwaysApply: false
---
- Import the library from src/ (development version, not
  pip-installed)
- Include a results summary cell at the end with key metrics
- Save experiment results as JSON alongside the notebook
- Always document: hypothesis, method, result, conclusion

```

Always-on rule · injected into every chat session:

```

---
description: Core project standards
alwaysApply: true
---
- Python 3.11+, type hints everywhere, Pydantic v2
- Linting: ruff. Formatting: ruff format. Never use flake8.
- Tests: pytest in tests/, run with pytest -v
- All LLM calls route through LiteLLM (not direct OpenAI/Anthropic
  clients)

```

What Good Rules Look Like

Rules should be **short** (under 50 lines), **specific** (not "write good code"), and **grounded** (reference real files with @filename). Think of them as the 10 things a new team member needs to know before touching a file.

Part 2: MCP Servers · Give the AI Access to Live Tools

Docs: cursor.com/docs/mcp

This is the single biggest productivity lever most developers ignore. MCP (Model Context Protocol) servers connect Cursor to external tools · documentation, GitHub, browsers, databases, web search · so the agent can use them directly instead of hallucinating answers.

The Problem MCPs Solve

LLM training data is frozen in time. When you ask about LangChain 0.3, Pydantic v2, or React 19, the agent may give you deprecated APIs, removed methods, or outdated patterns. You end up debugging the AI's suggestions instead of your code.

MCP servers fix this by giving the agent access to **live** data sources.

Setup

Create `.cursor/mcp.json` in your project root (shared with the team) or `~/ .cursor/mcp.json` in your home directory (personal).

The Essential Stack (All Free, No API Keys)

These are the MCPs I recommend for any Python + JS project. All are free.

`.cursor/mcp.json`:

```
{
  "mcpServers": {
    "context7": {
      "url": "https://mcp.context7.com/mcp"
    },
    "playwright": {
      "command": "npx",
      "args": ["-y", "@playwright/mcp@latest"]
    }
  }
}
```

Context7 · Live Documentation for 9,000+ Libraries

context7.com | Setup for Cursor

Context7 indexes source repos and serves current documentation on demand. When you ask about Pydantic, FastAPI, React, LiteLLM, Docusaurus · it fetches the real docs, not what the LLM memorized from 2023.

What changes:

| Without | With Context7 | |-----|-----| "Use model_validator" (deprecated Pydantic v1 API) |
Fetches current Pydantic v2 docs, gives correct @model_validator | | Guesses at LiteLLM provider strings |
Fetches the actual provider list from LiteLLM docs | | Wrong Docusaurus config options | Fetches current
Docusaurus 3.x configuration reference |

Usage: Add "use context7" to your prompt, or create a rule that triggers it automatically:

```

description: Auto-trigger Context7 for library documentation
alwaysApply: true
---
When I ask about library APIs, framework configuration, or need
code examples
for external packages, use the Context7 MCP to fetch current
documentation.
Do not rely on training data for library-specific syntax.

```

Optional: Free API key at context7.com/dashboard for higher rate limits.

Playwright · Browser Automation

Microsoft's official MCP for controlling browsers. The agent can navigate to your running app, take accessibility snapshots, click elements, fill forms, and report what it finds. Essential for any project with a web frontend.

LangChain & CrewAI Docs

If your project integrates with LangChain or CrewAI, dedicated MCP servers give the agent access to current documentation for both frameworks · helpful since their APIs change frequently between versions.

MCPs for Research and Discovery

| MCP | What It Does | Cost | How to Add | |-----|-----|-----|-----| | **Fetch** (Anthropic) | Downloads any webpage, converts to markdown | Free | Already built into Cursor | | **Tavily** | AI-optimized web search · great for finding papers and technical articles | Free tier | "url": "https://mcp.tavily.com/mcp" via `npx -y mcp-remote` | | **Perplexity** | AI search engine as a tool | Free tier | `npx -y @perplexity-ai/mcp-server` | | **Stack Overflow** | Search and read Stack Overflow threads | Free | `npx mcp-remote mcp.stackoverflow.com` | | **Markitdown** (Microsoft) | Convert PDF, DOCX, XLSX to markdown · useful for research papers | Free | See github.com/microsoft/markitdown |

MCPs for DevOps

| MCP | What It Does | Cost | How to Add | |-----|-----|-----|-----| | **GitHub** | Create PRs, search issues, read diffs · from chat | Free (needs PAT) | `npx -y @modelcontextprotocol/server-github + GITHUB_TOKEN` | | **Vercel** | Manage deployments | Free tier (OAuth) | "url": "https://mcp.vercel.com" | | **Linear** | Issue tracking | Free tier (OAuth) | "url": "https://mcp.linear.app/mcp" | | **Sentry** | Error tracking · ask what errors happened today? | Free tier (OAuth) | "url": "https://mcp.sentry.dev/mcp" |

MCPs for Documentation Lookup (All Free SSE Servers)

If you work with specific frameworks beyond what Context7 covers, these give direct access:

| MCP | URL | |-----|-----| | **Cloudflare** | <https://docs.mcp.cloudflare.com/mcp> | | **Svelte** | <https://mcp.svelte.dev/mcp> | | **Astro** | <https://mcp.docs.astro.build/mcp> | | **MS Learn** | <https://learn.microsoft.com/api/mcp> | | **Angular** | `npx -y @angular/cli mcp` | | **LiveKit** | <https://docs.livekit.io/mcp> |

These are all free SSE servers · just add the URL to `mcp.json` and they work.

How to Discover More

- **cursormcp.dev** · Directory of 1,500+ MCP servers, searchable by category
- **Cursor Marketplace** · One-click install for vetted servers
- **Settings · Features · MCPs** · Toggle servers on/off without removing them

Config Variables

Use `${env:GITHUB_TOKEN}`, `${workspaceFolder}`, `${userHome}` in your `mcp.json` values. Never hardcode secrets.

Part 3: Skills · Repeatable Workflows

Docs: cursor.com/docs/skills

A skill is a set of instructions that the agent can invoke as a single command. Think of it as a saved prompt with structure · it includes when to trigger, what steps to follow, and what scripts to run.

Skill Structure

A skill lives in its own directory under `.cursor/skills/` (or `~/.cursor/skills/` for personal cross-project skills). Each directory has a `SKILL.md` file with YAML frontmatter and step-by-step instructions.

Skills can also include `scripts/`, `references/`, and `assets/` subdirectories for executable code and reference materials.

```
.cursor/skills/
... code-reviewer/
.   ... SKILL.md
... write-blog-post/
.   ... SKILL.md
... deploy-staging/
.   ... SKILL.md
.   ... scripts/
.       ... validate-env.sh
```

Example: Code Review Skill

```
---
name: code-reviewer
description: >
  Review code changes for quality, security, and adherence to
  project conventions.
  Use when the user asks for a code review, PR review, or quality
  check.
---
# Code Reviewer
## Steps
1. Identify the changed files (use git diff or the user's
description)
2. For each file, read the corresponding rule (.cursor/rules/) to
know the conventions
3. Check for: type hints, error handling, test coverage, security
issues
4. Verify naming conventions and file organization match project
patterns
5. Run: `ruff check .` and `mypy --strict` on changed files
6. Report: issues found, severity, suggested fixes · grouped by file
```



```
## Reference
- See tests/conftest.py for test fixture patterns
- See .cursor/rules/ for per-workstream conventions
```

Example: Blog Post Skill

```
---
name: write-blog-post
description: >
  Write a blog post for the documentation site. Use when the user
  asks to
  create a new blog post, article, or content piece.
---
# Blog Post Writer
## Steps
1. Ask for the topic and target audience if not provided
2. Check existing posts for tone, format, and frontmatter style
3. Use Context7 MCP to fetch current API examples for any libraries
   mentioned
4. Write with proper frontmatter (title, authors, tags, date)
5. Include code examples that use real project imports · never
   pseudocode
6. Save to the blog directory with the naming convention:
   YYYY-MM-DD-slug.md
```

Invocation

- **Automatic:** Agent sees the description and applies when relevant
 - **Manual:** Type `/code-reviewer` or `/write-blog-post` in chat
 - **Slash-command only:** Add `disable-model-invocation: true` to frontmatter
-

Part 4: Subagents · Parallel Workers With Separate Context

Docs: cursor.com/docs/subagents

When the agent searches your codebase, runs shell commands, or controls a browser, it generates huge amounts of intermediate output (file contents, command logs, DOM snapshots). Subagents handle this in a separate context window, then return a summary.

Built-in Subagents (Already Active)

| Subagent | When It's Used | Why It Matters | |-----|-----|-----| | **Explore** | "Find all classes that implement the `process` method" | Runs 10+ parallel searches on a faster model. You see a summary, not raw `grep` output. | | **Bash** | "Run the test suite and tell me what failed" | Verbose `pytest` output stays in the subagent. You get pass/fail + specific failures. | | **Browser** | "Open `localhost:5173` and click through the onboarding" | DOM snapshots and screenshots stay isolated. You get a report of what happened. |

You don't configure these · they activate automatically.

Custom Subagents

Create `.cursor/agents/` and add a markdown file for each specialist.

Example · a post-edit verifier:

```
---
name: post-edit-verifier
description: >
  Validates code after changes. Use proactively when source files
  are modified.
  Runs tests, checks types, verifies the change didn't break
  anything.
model: fast
---
You verify code changes. When invoked:
1. Identify which files were modified
2. Run the relevant test suite: pytest -k <module_name> -v
3. Run type checking: mypy --strict on the changed files
4. Run linting: ruff check on the changed files
5. Check for common issues: missing imports, unused variables,
  broken interfaces
6. Report: pass/fail, specific issues, suggestions
Do not fix issues yourself · only report what you find.
```

The model: fast directive tells Cursor to use a cheaper, faster model for this subagent · it doesn't need deep reasoning, just systematic checking.

Foreground vs Background

- **Foreground** · Blocks until done. Use for "verify this before I continue."
- **Background** · Returns immediately. Use for "run the full test suite while I keep working."

Explicit Invocation

Type /post-edit-verifier or say "use the verifier subagent" in chat. You can also reference it in rules to trigger automatically.

Part 5: Hooks · Automated Guardrails

Docs: cursor.com/docs/hooks

Hooks run scripts before or after agent actions. They're the safety net · auto-formatting, blocking dangerous commands, auditing what the agent does.

.cursor/hooks.json

```
{
  "version": 1,
  "hooks": {
    "afterFileEdit": [
      {
        "command": ".cursor/hooks/format-python.sh",
        "matcher": "*.py"
      }
    ],
    "beforeShellExecution": [
```

```

    {
      "command": ".cursor/hooks/block-dangerous.sh",
      "matcher": "rm -rf|drop table|curl.*prod"
    }
  ]
}

```

`.cursor/hooks/format-python.sh` (make executable with `chmod +x`):

```

#!/bin/bash
cat > /dev/null # consume stdin
ruff check --fix . 2>/dev/null
ruff format . 2>/dev/null
exit 0

```

What Hooks Can Do

| Hook Event | Practical Use | |-----|-----| | `afterFileEdit` | Auto-run ruff/black on Python, Prettier on JS | | `beforeShellExecution` | Block `rm -rf`, `curl` to production URLs, `git push --force` | | `sessionStart` | Inject environment context (branch name, recent commits) | | `beforeMCPExecution` | Audit which MCP tools are being called | | `subagentStart` / `subagentStop` | Track subagent usage and cost |

Prompt-Based Hooks (No Script Needed)

For quick guardrails, use a prompt instead of a script:

```

{
  "hooks": {
    "beforeShellExecution": [
      {
        "type": "prompt",
        "prompt": "Is this shell command safe for a development environment? Block anything that modifies production data or deletes files recursively."
      }
    ]
  }
}

```

The LLM evaluates the condition and blocks if unsafe.

Part 6: Browser Automation - Test Your UI From Chat

Cursor has a built-in Browser subagent that uses MCP tools to control a browser. With the Playwright MCP installed, you can test your web app without writing test scripts.

What You Can Do

Ask the agent:

- "Open localhost:3000, log in with test credentials, and check if the Dashboard loads"
- "Navigate to the Settings page and verify all form fields render correctly"
- "Click through the checkout flow and report any console errors"

- "Take a screenshot of the signup page on mobile viewport"

How It Works

- 1 The agent uses Playwright to open a browser
- 2 It takes an accessibility snapshot (structured DOM, not raw HTML)
- 3 It interacts with elements by reference (click, type, scroll)
- 4 It returns a summary of what it found

This is especially powerful after UI changes · instead of manually clicking through flows, ask the agent to verify them. No test scripts to write, no Selenium boilerplate. Just describe what to check in plain English.

Part 7: GitHub Integration

With GitHub MCP (Recommended)

Add to `.cursor/mcp.json`:

```
{
  "mcpServers": {
    "github": {
      "command": "npx",
      "args": [ "-y", "@modelcontextprotocol/server-github" ],
      "env": {
        "GITHUB_PERSONAL_ACCESS_TOKEN": "${env:GITHUB_TOKEN}"
      }
    }
  }
}
```

Set `GITHUB_TOKEN` in your shell environment (not in the file).

What This Enables

- **"Create a PR for these changes"** · Agent writes a description, opens the PR
- **"What issues mention the payment flow?"** · Searches issues without leaving Cursor
- **"Summarize the comments on PR #42"** · Reads PR review comments
- **"Add a label to issue #15"** · Direct repo management

Without MCP

Cursor has built-in git support (commits, diffs, history). For GitHub-specific features (PRs, issues, reviews), you need the MCP server.

Part 8: Plugins · Share Everything as a Package

Docs: cursor.com/docs/plugins

A plugin bundles rules, skills, agents, MCP configs, and hooks into a single installable package. Anyone who installs it gets your entire setup.

When to Create a Plugin

- You want open-source contributors to get your project conventions automatically
- You want to share the same setup across multiple repos in your organization
- You want to publish to the Cursor Marketplace

Plugin Structure

```
your-cursor-plugin/
... .cursor-plugin/
.   ... plugin.json          # {"name": "your-plugin", "version":
"1.0.0"}
... rules/
.   ... backend.mdc
.   ... frontend.mdc
.   ... docs.mdc
... skills/
.   ... code-reviewer/
.       ... SKILL.md
... agents/
.   ... post-edit-verifier.md
... .mcp.json                # Context7, Playwright, etc.
... README.md
```

Submit at cursor.com/marketplace/publish.

Part 9: .cursor/settings.json and Plugins

What .cursor/settings.json Is

This file enables or disables **Cursor plugins** for your project. It is *not* VS Code settings (those live in `.vscode/settings.json`). You don't create it by hand - when you install a plugin, Cursor adds the entry automatically.

```
{
  "plugins": {
    "continual-learning": {
      "enabled": true
    }
  }
}
```

Each key under "plugins" is a plugin name, and "enabled": true/false toggles it for this project.

What Plugins Are

Plugins bundle rules, skills, agents, MCP servers, hooks, and commands into installable packages. You install them from the Cursor Marketplace or by typing `/add-plugin <name>` in Agent chat.

Once installed, the plugin's components become active in your project - its rules appear in your rules list, its skills become invocable, its MCP servers connect.

Useful Plugins to Know About

| Plugin | What It Does | Install | |-----|-----|-----| | **Continual Learning** | Mines your chat sessions, extracts recurring corrections, writes them to `AGENTS.md` automatically | `/add-plugin continual-learning` | | **Cursor Team Kit** | Internal workflows used by Cursor's own team · CI monitoring, PR creation, code review, merge conflicts | `/add-plugin cursor-team-kit` | | **Superpowers** | Core skills library · TDD, debugging, collaboration patterns | `/add-plugin superpowers` | | **Figma** | Figma MCP server + design workflow skills | From marketplace | | **Vercel** | React/Next.js best practices + deploy to Vercel | From marketplace | | **Stripe** | Stripe integration best practices + MCP server | From marketplace | | **Sentry** | Error tracking · debug production issues from chat | From marketplace | | **1Password** | Securely manage development secrets | From marketplace | | **Langfuse** | LLM tracing, prompt management, evaluation | From marketplace | | **Create Plugin** | Scaffolds a new Cursor plugin with manifest and quality checks | From marketplace |

Browse the full catalog at cursor.com/marketplace.

The Continual Learning Plugin (Recommended)

This one deserves special attention. It solves the problem of the agent forgetting your corrections between sessions.

How it works:

- 1 After each session, it scans your conversation for recurring corrections and workspace facts
- 2 It writes them as bullet points into an `AGENTS.md` file in your project root
- 3 Next session, the agent reads `AGENTS.md` and already knows your preferences

Example: You correct the agent three times that "we use ruff, not flake8." The plugin notices the pattern and adds –
`Use ruff for linting, not flake8` to `AGENTS.md`. Next session, the agent already knows.

When to use this vs rules:

- **Continual Learning** · You don't know what to write rules about yet. Let the plugin observe and build up `AGENTS.md` organically.
 - **Rules** · You already know your conventions. More powerful (glob scoping, per-file targeting).
 - **Both together** · The plugin discovers preferences, you promote the important ones to rules over time.
-

Best Practices (From Cursor's Team and Power Users)

These are drawn from Cursor's official best practices, the prompting docs, and real-world usage.

1. Plan Before You Code

For anything non-trivial, use **Plan Mode** (`Shift+Tab`). The agent researches your codebase, asks questions, and writes a plan. You approve before it writes code.

- Save plans with "Save to workspace" · stored in `cursor/plans/`
- If the agent builds the wrong thing, **revert and refine the plan** · don't try to fix through follow-up prompts
- Plans double as team documentation

2. Write Specific Prompts

Bad: "Add tests for auth"

Good: "Write a test case for the logout endpoint in `app/auth/router.py` covering the edge case when the session has expired. Use the patterns in `tests/unit/`. Don't mock the database · use the test fixtures in `conftest.py`."

Structure complex prompts as:

- **Goal:** One line · what you want
- **Context:** Current behavior vs. desired behavior
- **Constraints:** What NOT to change
- **Acceptance criteria:** How to know it's done

3. Let the Agent Find Context

You don't need to @-mention every file. The agent has semantic search and grep · it finds relevant files automatically. Only use @filename when you know the exact file.

Including irrelevant files confuses the agent about what matters.

4. Know When to Start a New Chat

Start fresh when:

- You finished a logical unit of work
- The agent keeps making the same mistake
- You're switching tasks

Continue when:

- You're debugging what it just built
- You're iterating on the same feature

Long conversations accumulate noise. If the agent gets less effective, start over.

5. Reference Past Work (Don't Re-explain)

Use @Past Chats to point the agent at a previous conversation instead of re-typing everything.

6. Provide Verifiable Goals

Agents can't fix what they can't check. Give them:

- **Tests:** "Keep iterating until all tests pass"
- **Type checking:** "Run mypy after changes"
- **Linting:** "Run ruff check . and fix any issues"

The best workflow: write tests first (or ask the agent to), then ask it to implement until tests pass.

7. Use Checkpoints for Safety

The agent creates checkpoints before major edits. If it goes wrong, click any checkpoint in the chat to preview and restore that state. This is separate from git · use git for permanent version control, checkpoints for quick undo.

8. Run Agents in Parallel

For hard problems, run the same prompt on multiple models simultaneously (select multiple from the model dropdown). Compare results side by side.

For independent tasks, use git worktrees · each agent gets its own isolated copy of the codebase.

9. Start Simple, Add Rules Gradually

Don't write 20 rules on day one. Start with one always-apply rule for core standards. Add more only when you notice the agent making the same mistake repeatedly.

10. Use Images

Paste screenshots directly into chat:

- Design mockups · "Implement this UI"
 - Error screenshots · "Debug this"
 - Console errors · faster than typing the stack trace
-

What a Complete `.cursor/` Setup Looks Like

Here's the full file tree of a production-ready Cursor configuration, with every file explained. This is what you're building toward.

```
your-project/
├── ... .cursor/
│   ├── ... rules/
│   │   ├── ... core-standards.mdc          # Always-on: Python/JS standards,
│   │   ├── testing, imports
│   │   ├── ... sdk-library.mdc             # Glob: src/**/*.py · SDK patterns,
│   │   ├── class hierarchy
│   │   ├── ... app-backend.mdc             # Glob: app/api/**/*.py · FastAPI,
│   │   ├── auth, DB conventions
│   │   ├── ... app-frontend.mdc            # Glob:
│   │   ├── app/web/src/**/*.{jsx,tsx} · React, routing, CSS
│   │   ├── ... website-docs.mdc            # Glob: website/docs/**/*.md ·
│   │   ├── Docusaurus, frontmatter
│   │   ├── ... research-notebooks.mdc      # Glob: docs/**/*.ipynb · notebook
│   │   ├── conventions, metrics
│   │   ├── .
│   │   ├── ... skills/
│   │   │   ├── ... code-reviewer/
│   │   │   │   ├── ... SKILL.md            # Multi-step: read changes · check
│   │   │   │   ├── conventions · run lints · report
│   │   │   │   ├── ... write-blog-post/
│   │   │   │   ├── ... SKILL.md            # Multi-step: pick topic · fetch
│   │   │   ├── docs · write · save
│   │   │   ├── .
│   │   │   ├── ... agents/
│   │   │   │   ├── ... post-edit-verifier.md # Fast subagent: runs tests +
│   │   │   │   ├── types + lint after changes
│   │   │   │   ├── .
│   │   │   │   ├── ... hooks/
│   │   │   │   │   ├── ... format-python.sh # Auto-runs ruff check --fix &&
│   │   │   │   │   ├── ruff format after edits
│   │   │   │   │   ├── ... block-dangerous.sh # Blocks rm -rf /, drop table,
│   │   │   │   │   ├── curl to prod URLs
│   │   │   │   │   ├── .
│   │   │   │   │   ├── ... mcp.json         # Context7 + Playwright +
│   │   │   │   ├── GitHub (+ optional research MCPs)
```



```

·    ... hooks.json                    # Maps hook scripts to file
patterns and events
·    ... settings.json                # Plugin toggles (e.g.,
continual-learning: enabled)
·
... AGENTS.md                        # Root-level instructions ·
simpler alternative to rules
... ..

```

What Each Layer Does

| Layer | Files | Effect | |-----|-----|-----| | **Rules** | 6 .mdc files | Agent automatically knows conventions for whichever file you're editing. No re-explaining. | | **Skills** | 2 SKILL.md files | Complex workflows become one-command operations. Agent follows the steps you defined. | | **Subagents** | 1 .md agent | Verification runs in parallel on a fast model. You keep working while it checks. | | **Hooks** | 2 shell scripts | Every Python file auto-formatted on save. Dangerous commands blocked before execution. | | **MCP servers** | 3-5 servers | Live docs (Context7), browser testing (Playwright), repo management (GitHub). | | **Plugins** | settings.json | Continual Learning mines your corrections and writes them to AGENTS.md automatically. | | **AGENTS.md** | 1 markdown file | Fallback instructions for agents that don't read .cursor/ · also useful for non-Cursor contributors. |

The Order to Build It

You don't need everything on day one. Build in layers:

| When | What to add | Time | Impact | |-----|-----|-----| | **Day 1** | mcp.json (Context7 only) + 1 always-on rule | 5 min | Agent stops hallucinating library APIs | | **Week 1** | Add Playwright MCP + per-workstream rules | 30 min | Agent knows your conventions per area | | **Week 2** | Add 1 skill + 1 subagent + hooks | 30 min | Repeatable workflows + auto-formatting | | **Week 3** | Add GitHub MCP + research MCPs + AGENTS.md | 20 min | Full workflow · PRs, research, docs from chat | | **Month 2** | Install Continual Learning plugin | 2 min | Agent learns from your corrections automatically | | **Optional** | Package as a Cursor plugin | 1 hour | Share with your team or the open-source community |

How It All Works Together · A Real Session

You've seen every file. Now watch them work as a system. This section walks through what actually happens from the moment you open Cursor to the end of a productive session.

Step 1: You Open the Project

```
cursor .
```

Before you type a single prompt, Cursor scans .cursor/ and loads everything:

What loads and when

| What Cursor loads | When | How you verify | |---|---|---| | settings.json | Immediately | Continual Learning plugin activates silently | | rules/* .mdc | Immediately | **Settings · Cursor · Rules** · all 6 listed with their glob patterns | | mcp.json | Immediately | **Settings · Features · MCP** server shows green (connected) or red (failed) | | hooks.json | Immediately | Active but invisible until triggered | | skills/ | Indexed on open | Type / in chat · your skills appear in the slash-command menu | | agents/ | Indexed on open | Available when the agent delegates, or when you invoke explicitly | | AGENTS.md | Read at chat start | Injected into every new conversation automatically |

Everything is ready. You haven't configured anything · it was all in the committed `cursor/` folder.

Step 2: You Open a File · The Right Rules Activate

Say you open `src/templates/step_reasoner.py` and type:

```
Improve this template's prompt quality
```

Behind the scenes: rule matching

Cursor checks every rule's globs against the open file:

```
| Rule | Glob | Match? | |-----|-----|-----| | core-standards.mdc | (always apply) | Yes · always  
injected | | sdk-library.mdc | src/**/*.py | Yes · file is in src/ | | app-backend.mdc |  
app/api/**/*.py | No | | app-frontend.mdc | app/web/src/**/*.{jsx,tsx} | No |  
| website-docs.mdc | website/docs/**/*.md | No | | research-notebooks.mdc |  
docs/**/*.ipynb | No |
```

Result: **2 rules injected**, 4 skipped.

The agent now knows · without you saying anything · that this file follows SDK conventions, uses Pydantic, needs type hints, and should be linted with ruff. You just say "improve this" and the context is already there.

If you switch to `app/web/src/pages/Dashboard.jsx` · the SDK rule deactivates and the frontend rule activates instead. Same agent, different conventions, zero re-explanation.

You can see which rules are active by checking the **context pills** at the top of the chat window.

Step 3: You Ask About a Library · MCP Fetches Live Docs

```
Update the LiteLLM integration to use the latest provider format.  
Use context7.
```

Behind the scenes: MCP call

- 1 Agent sees "use context7" (or your always-on rule tells it to check)
- 2 It calls the **Context7 MCP server** over the network
- 3 Context7 fetches the *current* LiteLLM documentation from source
- 4 Agent receives the real, up-to-date API · not what it memorized from training data
- 5 It writes code using the correct, current syntax

Without MCP: Agent guesses based on training data (possibly 1-2 years old). You debug its hallucinations.

With MCP: Agent fetches live docs. You get correct code on the first try.

MCP tool calls appear as collapsible sections in the chat · you can expand them to see exactly what was fetched.

To check MCP status at any time: Settings · Features · MCP. Green dot = connected. Red dot = failed (click for error details). Toggle servers on/off without deleting them.

Step 4: The Agent Edits a File · Hooks Auto-Format

The agent just edited `src/utils/helpers.py`. What happens next:

Behind the scenes: hook execution

- 1 The edit completes
- 2 `hooks.json` has an `afterFileEdit` hook matching `*.py`

- 3 Cursor runs `.cursor/hooks/format-python.sh`
- 4 The script executes `ruff check --fix .` and `ruff format .`
- 5 The file is now auto-formatted

You see a small notification: *"Hook executed: format-python.sh"*

You never run the formatter manually. Every Python file the agent touches is auto-formatted.

The safety hook works the same way. If the agent tries to run `rm -rf /` or `curl` to a production URL, the `beforeShellExecution` hook intercepts it:

Blocked command

BLOCKED: Dangerous command detected · `rm -rf /`

The command never executes. The agent gets the error and tries a safer approach.

Step 5: You Invoke a Skill · The Agent Follows Your Playbook

Type in chat:

`/code-reviewer`

Or just describe the task naturally · "review the changes I made to the auth module." The agent reads the skill description and decides to use it automatically.

Behind the scenes: skill execution

- 1 Agent loads `SKILL.md` from `.cursor/skills/code-reviewer/`
- 2 Follows each step in order:
 - Read changed files
 - Check conventions from matching rules
 - Run linting and type checking
 - Report issues grouped by file
- 3 You see each step executing in the chat as it progresses

Without a skill: Agent improvises a different review process each time.

With a skill: Same thorough workflow, every time, no matter who runs it.

Step 6: A Subagent Runs in Parallel

After the agent makes changes, it decides the verifier should check things (or you say "verify this"):

Behind the scenes: subagent delegation

- 1 A new lightweight agent instance spawns with its **own context window**
- 2 It reads `post-edit-verifier.md` for instructions
- 3 It runs tests, checks types, runs linting · all in the background
- 4 Verbose output (full pytest log, mypy trace) stays in the subagent's context
- 5 When done, it sends a **summary** back to the main agent

You see: *"Verifier passed: 14 tests pass, types clean, no lint issues"*

You don't see: 200 lines of pytest output.

The key insight: subagents keep your main chat clean. The main agent stays focused on your task while verification happens in parallel.

Step 7: Continual Learning Observes (Silently)

Because `settings.json` has `continual-learning` enabled, the plugin watches your conversations:

Behind the scenes: learning from corrections

- 1 During your session, you correct the agent: *"No, use ruff not flake8"*
- 2 This happens again in another session
- 3 The plugin detects the pattern
- 4 It writes to `AGENTS.md`: - Use ruff for linting, not flake8
- 5 **Next session**, the agent reads `AGENTS.md` on start · it already knows

You never make that correction again.

Over weeks, `AGENTS.md` grows organically into a knowledge base of your preferences. You can promote the important ones to proper rules anytime.

A Typical Day With This Setup

```
Morning · SDK work:
  Open Cursor · everything loads automatically (2 seconds)
  Open a library file · SDK rule activates
  "Improve this class" · agent knows the patterns, follows the skill
  Agent edits .py · hook auto-formats with ruff (instant)
  "Run the tests" · subagent runs pytest in background, reports
pass/fail
Afternoon · Frontend work:
  Open Dashboard.jsx · frontend rule activates, SDK rule deactivates
  "Add a loading spinner to the data table" · agent knows React
conventions
  "Test it" · Playwright MCP opens localhost, clicks through the
flow, reports
End of day · Ship it:
  "Create a PR for today's changes" · GitHub MCP creates the PR with
description
  "Write a blog post about the improvements" · blog skill activates
  Agent uses Context7 to fetch current Docusaurus frontmatter
format
  Hook auto-formats. Subagent verifies. PR is ready for review.
```

No conventions re-explained. No manual formatting. No browser switching. No hallucinated APIs.

Quick verification prompt

Open any new chat and type:

```
What rules are active? What MCP servers are connected?
What skills are available?
```

The agent lists everything it can see. That tells you your setup is working.

Sharing Your Setup

| Method | Who It's For | How | |-----|-----|-----| | **Commit** `.cursor/` **to git** | Anyone who clones the repo | They get rules, MCPs, hooks, skills automatically | | **Add** `AGENTS.md` **to root** | Contributors who don't use Cursor | Readable instructions that also work in other AI tools | | **Create a Cursor plugin** | Open-source community | One-click install from the marketplace | | **Link from your README** | Everyone | "We use Cursor with these conventions" |

The AGENTS.md Alternative

If rules feel like too much setup, create a single `AGENTS.md` file in your project root. It's plain markdown · no frontmatter, no globs. The agent reads it automatically.

```
# Project Instructions
## Code Style
- Python: Pydantic v2, ruff, mypy strict. Tests: pytest.
- React: functional components, CSS variables, useAuth() for auth
- All LLM calls go through LiteLLM, never direct provider clients
## Architecture
- Library source: src/ · pip-installable SDK
- Backend: app/api/ (routers), app/services/ (logic), app/db/
(models)
- Frontend: app/web/src/ · React + react-router-dom
- Docs site: website/ · Docusaurus
- Research: docs/, notebooks/ · Jupyter experiments
## Important
- Never commit .env files or API keys
- Run ruff check . before committing
- When in doubt, check tests/conftest.py for test fixture patterns
```

You can also nest `AGENTS.md` in subdirectories `·app/AGENTS.md`, `website/AGENTS.md` · for area-specific instructions. The agent reads all of them.

Rules vs AGENTS.md:

- Rules are more powerful (glob scoping, auto-apply, Apply Intelligently)
- `AGENTS.md` is simpler (just markdown) and works in other AI tools too
- You can use both · they combine, with rules taking precedence

Reference

Configuration Files

| File | Purpose | Docs | |-----|-----|-----| | `.cursor/rules/*.mdc` | Persistent AI guidance, scoped by file pattern | cursor.com/docs/rules | | `.cursor/skills/*SKILL.md` | Repeatable workflows the agent can invoke | cursor.com/docs/skills | | `.cursor/agents/*.md` | Custom subagents with separate context windows | cursor.com/docs/subagents | | `.cursor/mcp.json` | External tools (docs, GitHub, browser, etc.) | cursor.com/docs/mcp | | `.cursor/hooks.json` | Scripts that run before/after agent actions | cursor.com/docs/hooks | | `.cursor/settings.json` | Plugin configuration (e.g., continual-learning) | cursor.com/docs/plugins | | `AGENTS.md` | Simple markdown instructions (alternative to rules) | cursor.com/docs/rules |

Essential Links

| Resource | URL | |-----|-----| | Cursor Docs Sitemap | cursor.com/llms.txt | | Best Practices Blog | cursor.com/blog/agent-best-practices | | MCP Directory (1,500+ servers) | cursormcp.dev | | Cursor Marketplace (plugins) | cursor.com/marketplace | | Context7 Setup for Cursor | context7.com/docs/clients/cursor | | Agent Skills Standard | agentskills.io | | Prompting Guide | cursor.com/docs/agent/prompting | | Plan Mode | cursor.com/docs/agent/plan-mode |

Closing Thoughts

Most developers use Cursor as a smarter autocomplete. They type a prompt, get a response, and repeat · re-explaining their project conventions every session, debugging hallucinated APIs, and manually running formatters after every edit.

The developers who get 10x value from Cursor are the ones who spend an hour configuring it. Rules eliminate re-explanation. MCP servers eliminate hallucination. Hooks eliminate manual formatting. Skills eliminate re-inventing workflows. Subagents eliminate context overload. Plugins make all of it shareable.

The entire setup lives in a `.cursor/` folder that you commit to git. Every contributor who clones the repo inherits it. No onboarding doc to read, no setup script to run. The AI just *knows* how your project works.

That's the difference between using Cursor and *using* Cursor.

Ideated and co-written with Cursor by Dhiraj Pokhrel. Published by Sadhira AI. Built while developing mycontext-ai, a cognitive template SDK for LLMs. If you found this useful, star the repo or share the guide.