

# Les 9 — Cursor Deep Dive

Kennisdocument — Van Agent Harness tot Code Review

**Vak:** AI-Assisted Development

**Opleiding:** NOVI Hogeschool Utrecht

**Docent:** Tim

**Tech stack:** Next.js 16, TypeScript, Tailwind CSS

## Leerdoelen

Na het lezen van dit document kun je:

- Uitleggen hoe een coding agent werkt (harness: instructions + tools + model)
- `.cursor/rules` opzetten met meerdere bestanden en YAML frontmatter
- Plan Mode gebruiken om een feature te plannen voordat je bouwt
- Agent Mode inzetten voor multi-file feature development
- Debug Mode toepassen voor systematisch debuggen
- Code review doen, tests schrijven, en semantic commits maken

## Inhoud

1. Het Agent Harness — hoe coding agents werken
2. Rules & Skills — de `.cursor/rules` directory
3. Context Management — @ mentions en werkgeheugen
4. Plan Mode — eerst denken, dan bouwen
5. Agent Mode & Ask Mode — uitvoeren vs. vragen
6. Debug Mode — systematisch bugs opsporen
7. Code Review & Testing — kwaliteit bewaken
8. Semantic Commits — je werk netjes opleveren

## 1

## Het Agent Harness

Een coding agent is meer dan een chatbot. Het is een AI-model dat draait binnen een **harness** — een systeem dat het model aanstuurt en uitrust met gereedschap. Cursor is zo'n harness. Wanneer je met Cursor werkt, communiceer je niet rechtstreeks met het AI-model. Het harness zit ertussen en regelt alles.

### De drie onderdelen

Onderdeel	Wat het doet	Jouw rol
<b>Instructions</b>	System prompt + jouw rules-bestanden. Vertelt de agent HOE hij moet werken.	Je schrijft <code>.cursor/rules</code> bestanden
<b>Tools</b>	Bestanden lezen/bewerken, door code zoeken, terminal commando's uitvoeren.	Je geeft toestemming (Accept/Reject)
<b>Model</b>	Het AI-model dat beslissingen neemt (Claude, GPT-4o, etc.).	Je kiest het model in Cursor settings

### De Agent Loop

De agent werkt in een loop: **denken** → **actie uitvoeren** → **resultaat observeren** → **herhalen**. Net zoals jij als developer zou werken. De agent leest je code, bedenkt wat er moet gebeuren, voert een actie uit (bestand bewerken, commando draaien), bekijkt het resultaat, en herhaalt tot de taak klaar is.

**Belangrijk:** Het harness regelt de loop, de tools en de system prompt. Jij focust je op twee dingen: goede **instructions** (rules) en goede **prompts**.

## 2

## Rules & Skills

In Les 3 heb je `.cursorrules` gebruikt — één bestand in je project root. Cursor heeft dit systeem uitgebreid naar een **`.cursor/rules/` directory** met meerdere bestanden, elk voor een ander doel.

### De drie types rules

Type	Hoe activeren?	Voorbeeld
<b>Always-on</b>	<code>alwaysApply: true</code>	<code>project.mdc</code> — tech stack, conventies
<b>Auto-attached</b>	<code>globs: "src/**/*.tsx"</code>	<code>components.mdc</code> — component regels

Type	Hoe activeren?	Voorbeeld
Agent-requested	Via de <code>description</code> in frontmatter	<code>api-design.mdc</code> — agent vraagt het zelf op

## YAML frontmatter

Elk `.mdc` bestand begint met YAML frontmatter — metadata tussen twee rijen van drie streepjes (`---`):

```
# .cursor/rules/project.mdc
---
description: Algemene projectregels voor LinkVault
globs:
  alwaysApply: true
---

# LinkVault – Projectregels

## Tech Stack
- Next.js 16 met App Router
- TypeScript in strict mode
- Tailwind CSS voor alle styling
- In-memory data opslag (GEEN database)

## Conventies
- Code in het Engels, comments in het Nederlands
- Named exports, geen default exports
- Geen extra dependencies zonder toestemming
```

**Vuistregel:** houd rules kort en specifiek. Meer regels = meer ruis = slechtere output. Geef concrete voorbeelden in je rules — de agent leert van patronen, niet van abstracte beschrijvingen.

## Globs voor auto-attached rules

Met `globs` bepaal je wanneer een rule automatisch wordt geladen. Enkele voorbeelden:

Glob patroon	Matcht
<code>src/components/**/*.tsx</code>	Alle TSX bestanden in components (recursief)
<code>**/*.css, **/*.tsx</code>	Alle CSS en TSX bestanden in het hele project
<code>src/app/**/*.route.ts</code>	Alle API route bestanden

**Skills** zijn dynamische rules die Cursor automatisch kan laden (bijv. via `@docs` of `@web`). Skills zijn krachtig maar vallen buiten de scope van deze les.

## 3 Context Management

De agent heeft een **context window** — een beperkt werkgeheugen. Alles wat je toevoegt kost ruimte. Elk gesprek begint "leeg": de agent weet alleen wat je hem vertelt. Denk aan de agent als een briljante developer die net op je project begint. Hij kent de taal en frameworks, maar JOUW code kent hij niet.

### @ Mentions

Mention	Wat het doet	Wanneer gebruiken
<code>@file</code>	Voegt een specifiek bestand toe als context	Als je weet welk bestand relevant is
<code>@folder</code>	Voegt een hele map toe	Voor een overzicht van een module
<code>@codebase</code>	Cursor doorzoekt je hele project	Als je niet weet waar iets staat
<code>@web</code>	Zoekt op internet	Voor documentatie of actuele info
<code>@docs</code>	Doorzoekt specifieke documentatie	Voor framework-specifieke vragen
<code>@git</code>	Referentie naar git diff/history	Om recente changes te bespreken

### Gouden regels voor context

- **Nieuwe feature = nieuwe chat.** Dit is de allerbelangrijkste tip. Na teveel heen-en-weer raakt de context vervuild en gaat de agent afdwalen.
- **Combineer @ mentions** voor maximale relevantie: `@data.ts @BookmarkCard.tsx` — voeg een edit knop toe
- **Geef relevante context, niet alles.** Teveel context is net zo slecht als te weinig.
- **Sub-agents:** als je `@codebase` gebruikt, stuurt Cursor een sub-agent om door je code te zoeken. Die filtert en geeft alleen het relevante terug.

## 4 Plan Mode

**Start ELKE feature met een plan. Dit is de belangrijkste workflow die je leert.**

In Plan Mode denkt de agent na zonder code te schrijven. Hij doet research, stelt vragen, en maakt een stap-voor-stap plan in markdown. Dat plan kun je bewerken voordat de agent begint te bouwen.

## Workflow

- 1 **Ctrl+L** — Open een nieuwe chat
- 2 **Shift+Tab** — Schakel naar Plan Mode
- 3 **Typ je prompt** — Beschrijf WAT je wilt bouwen, met features en structuur
- 4 **Agent plant** — De agent onderzoekt, stelt vragen, maakt een plan
- 5 **Jij reviewt** — Lees het plan, pas aan waar nodig, schrap overbodige stappen
- 6 **Klik "Build"** — Schakel naar Agent Mode en laat het plan uitvoeren
- 7 **Review output** — Controleer de gegenereerde code

## Waarom Plan Mode?

Zonder plan gaat de agent z'n eigen gang. Dan ben je uren bezig met corrigeren. Een goed plan levert gerichte output, minder iteraties, en minder verspilde requests. Jij bent de architect, de agent is de aannemer.

## Voorbeeld prompt

```
Ik wil een bookmark manager app bouwen genaamd LinkVault.
```

```
Features:
```

- Bookmarks opslaan met URL, titel en tags
- Lijst van alle bookmarks in een card layout
- Filteren op tag (klikbare badges)
- Bookmarks bewerken (inline editing)
- In-memory data opslag in een apart data.ts bestand

```
Technische eisen:
```

- Next.js 16 met App Router, TypeScript, Tailwind CSS
- Server Components waar mogelijk

```
Welke stappen stel je voor om dit te bouwen?
```

*Let op: de prompt eindigt met een vraag ("welke stappen stel je voor?"), niet met een opdracht. Dat is Plan Mode — de agent denkt na, jij beslist.*

## 5

## Agent Mode & Ask Mode

Cursor heeft twee modes in de chat. Het verschil is simpel:

	Agent Mode	Ask Mode
<b>Wat doet het?</b>	Voert uit: bestanden bewerken, terminal draaien, nieuwe files maken	Antwoordt alleen: uitleg, vragen beantwoorden
<b>Wanneer?</b>	Features bouwen, refactoring, bug fixen, tests schrijven	Code begrijpen, concepten leren, architectuurbeslissingen
<b>Shortcut</b>	Tab (in chat)	Tab Tab (in chat)

**Vuistregel:** "Moet er code veranderen?" → Agent Mode. "Wil ik iets begrijpen?" → Ask Mode.

### Tips voor Agent Mode

- **Geef context mee** met @ mentions. Verwijs naar de bestanden die de agent nodig heeft.
- **Wees specifiek.** "Voeg een edit knop toe" is beter dan "verbeter de app".
- **Lees voordat je Accept klikt.** Scan de code op structuur, imports en logica.
- **Itereer in dezelfde chat** voor kleine aanpassingen. Nieuwe feature = nieuwe chat.

## 6

## Debug Mode

Debug Mode is geen knop maar een **aanpak**. Bij simpele bugs plak je de error in de chat. Maar bij complexe bugs heb je een systematische methode nodig.

### Simpel vs. Systematisch

Simpele bug	Complexe bug (Debug Mode)
1. Kopieer de error	1. Beschrijf het probleem met context
2. Plak in chat	2. Agent genereert hypotheses
3. Agent fixt het	3. Agent voegt logging toe
	4. Jij reproduceert de bug
	5. Agent analyseert de logs

Simpele bug	Complexe bug (Debug Mode)
	6. Agent stelt een gerichte fix voor

## Voorbeeld: de toUpperCase-bug

In de les heeft Tim handmatig een bug geïntroduceerd: `tag.toUpperCase()` in de filter-logica. Tags worden opgeslagen als "react" (lowercase), maar gefilterd als "REACT" (uppercase). Resultaat: de filter vindt niets.

### Debug prompt

Bug: Als ik filter op een tag, krijg ik geen resultaten.  
Maar er zijn wel bookmarks met die tag.

Stappen om te reproduceren:

1. De app heeft bookmarks met tags zoals "react"
2. Ik klik op de "react" tag badge om te filteren
3. Verwacht: bookmarks met tag "react" worden getoond
4. Werkelijk: geen resultaten, lege lijst

Onderzoek dit systematisch:

1. Genereer hypotheses over wat het probleem kan zijn
2. Voeg `console.log` toe om de filter logica te debuggen

Relevante bestanden: `@src/lib/data.ts` `@src/components/TagFilter.tsx`

**Onthoud:** als je de voorgestelde fix niet begrijpt, kun je hem niet valideren. Debug Mode helpt je om te BEGRIJPEN wat er mis is. Dat maakt je een betere developer.

## 7

## Code Review & Testing

Dezelfde standaard geldt voor AI-geschreven code als voor handgeschreven code. Voordat je iemand anders vraagt om je code te reviewen, doe je altijd eerst een self-review.

### Self-review met Find Issues

Cursor heeft een **Find Issues** feature die je code scant op mogelijke problemen. Je kunt ook de agent vragen om een gerichte review:

```
Review de code van mijn project. Kijk naar:
- TypeScript type safety (zijn er any types?)
- Error handling (wat als input ongeldig is?)
- Accessibility (aria labels, keyboard navigatie)
- Consistentie met onze .cursor/rules

@codebase

Geef een lijst van issues gesorteerd op prioriteit.
```

### Tests schrijven met de agent

Tests zijn je vangnet. Als je de agent later meer autonomie geeft, zorgen tests ervoor dat hij niks breekt. Vraag de agent om tests te schrijven:

```
Schrijf unit tests voor de functies in @src/lib/data.ts.

Test scenarios:
1. addBookmark - controleer dat een bookmark wordt toegevoegd
2. deleteBookmark - controleer dat een bookmark wordt verwijderd
3. getBookmarksByTag - controleer dat filtering correct werkt
4. Edge case: filteren op een tag die niet bestaat

Gebruik Vitest.
```

## 8

## Semantic Commits

In plaats van een grote commit met "app gebouwd", verdeel je je wijzigingen in logische groepen met een prefix die het type wijziging aangeeft:

Prefix	Betekenis	Voorbeeld
feat:	Nieuwe feature	feat: add bookmark creation form



Prefix	Betekenis	Voorbeeld
fix:	Bug fix	fix: resolve case sensitivity in tag filter
refactor:	Code verbetering	refactor: extract tag logic to separate util
test:	Tests toevoegen	test: add unit tests for bookmark CRUD
docs:	Documentatie	docs: add README with setup instructions
chore:	Config, dependencies	chore: add Cursor rules for project

## De realistische workflow

Je hoeft niet na elke stap te committen. In de praktijk bouw je vaak een tijdje door zonder te committen — je bent gefocust, je zit in de flow. Dat is prima.

Aan het eind heb je een grote uncommitted diff. Dan vraag je Cursor om die diff op te splitsen in nette, logische commits:

```
Bekijk mijn staged changes en maak semantische commits.
Verdeel in logische groepen:
- Data model en types
- UI componenten
- Tag filtering feature
- Bug fix (case sensitivity)
- Tests

Gebruik conventional commit format.
Maak de commits in een logische volgorde.
```

**Resultaat:** een schone commit history die het verhaal van je project vertelt. Elke commit beschrijft WAT er is veranderd en WAAROM. Dat is hoe professionele developers werken.

## Samenvatting: 5 Key Takeaways

### Rules

Maak meerdere `.cursor/rules` bestanden met YAML frontmatter. Houd ze kort, specifiek, en geef concrete voorbeelden.

### Plan Mode

Begin **ALTIJD** met een plan. Shift+Tab, beschrijf wat je wilt, review het plan, pas aan, en dan pas bouwen.

### Context

Geef de agent de juiste context met @ mentions. Nieuwe feature = nieuwe chat. Combineer mentions voor relevantie.

### Debug Mode

Voor complexe bugs: niet gokken, maar systematisch. Hypotheses → logging → reproduceren → analyseren → fixen.

### Code Review

Dezelfde standaard voor AI-code als voor handgeschreven code. Tests schrijven, self-review, semantic commits.

## Keyboard Shortcuts

Actie	Shortcut
Chat openen	Ctrl+L / Cmd+L
Plan Mode toggle	Shift+Tab
Agent Mode	Tab (in chat)
Ask Mode	Tab Tab (in chat)
Inline Edit	Ctrl+K / Cmd+K

Actie	Shortcut
Composer	Ctrl+I / Cmd+I
Autocomplete accepteren	Tab
Browser DevTools	F12

**Dit document bevat de kernstof van Les 9. Gebruik het als naslagwerk bij het maken van het huiswerk en bij toekomstige projecten. Vragen? Stel ze via het les-kanaal.**