Contextvenster en regels in Cursor: Hoe werkt het en hoe blijven regels behouden?

Inleiding:

Cursor is een AI-ondersteunde code-editor die grote language models (LLM's) integreert om de ontwikkelaar te assisteren. Een belangrijk aspect hiervan is hoe Cursor omgaat met zijn *context window* (de hoeveelheid tekst die het model in een sessie kan "onthouden") en de *regels* die je aan het model meegeeft. We onderzoeken hoe Cursor context en regels beheert op verschillende niveaus – van globale instellingen tot project-specifieke .cursorrules – en hoe modellen als Claude 3.5 "Sonnet" hierin passen. We kijken ook of regels na verloop van tijd uit de context verdwijnen (FIFO-probleem) en hoe je kunt zorgen dat instructies persistent en consistent blijven. Tot slot bieden we aanbevelingen op basis van recente inzichten in Cursor en vergelijkbare AI-contexttechnieken.

Beheer van het context window in Cursor

Grootte en gebruik van het contextvenster:

Cursor hanteert een limiet op de hoeveelheid tekst (in tokens) die het in één keer aan het model doorgeeft. In de standaard modus ("cmd-K" chat of de code-compositie) is dit rond de 10.000 tokens . Deze beperking is ingesteld om de responstijd (TTFT – *time to first token*) en output-kwaliteit in balans te houden . Als je meer context nodig hebt, biedt Cursor een *Long-Context Chat* modus die de maximale contextgrootte van het gekozen model gebruikt . Met Long-Context Chat kan bijvoorbeeld een model als Claude de volle 100k tokens context benutten (of GPT-4 de 32k tokens, etc.), wat handig is bij zeer grote codebases of uitgebreide gesprekken.

Samenstelling van de context:

Cursor stelt bij elke prompt een context samen die doorgaans bestaat uit: (a) eventuele **systeem- of regelprompt** (zoals globale/projectregels), (b) de relevante **code context** (bijv. de geopende file of functionele snippets die relevant zijn voor je vraag), en (c) de **chatgeschiedenis** tot dusver (binnen de tokenlimiet). Cursor voegt bijvoorbeeld automatisch de inhoud van je huidige bestand toe aan de prompt wanneer je daar vragen over stelt, zodat het model de juiste context heeft voor codebewerking . Bij gebruik van de chatfunctie heeft het model toegang tot alles wat in dat gesprek is besproken (tot de limiet) . Om binnen de 10k tokens te blijven hanteert Cursor een *sliding window*: als het gesprek langdurig is, worden oudere berichten en context weggelaten of samengevat zodra de limiet wordt overschreden (FIFO-principe). Dit betekent dat het **eerste dat in de context zat, er als eerste uit valt** als er nieuwe context bij komt en de limiet bereikt wordt. Belangrijke details van eerder in het gesprek kunnen dus verdwijnen tenzij Cursor of de gebruiker ze expliciet opnieuw toevoegt.

Contextbeheer tijdens langere sessies:

Cursor probeert relevante context te behouden door bijvoorbeeld oudere niet-relevante stukken weg te laten of samen te vatten. Toch is het mogelijk dat na veel interacties oorspronkelijke instructies of delen van de codecontext niet langer expliciet in het promptvenster staan. Sommige gebruikers hebben creatieve oplossingen hiervoor toegepast. Zo beschrijft iemand een experiment waarbij hij automatisch een uitgebreide SPEC.md genereert met daarin de architectuur en belangrijke details van het project, om deze als samenvatting in de context te houden . Dit soort *samenvattingsbestanden* of documentatie dienen als een vaste referentie zodat de kerninformatie niet verloren gaat, zelfs als de conversatie vordert. In feite is dit een vorm van contextmanagement buiten het directe geheugen: de essentie van het project blijft bewaard in beknopte vorm en kan telkens opnieuw in de prompt worden meegenomen.

Lokale modellen en context:

Cursor is primair ontworpen voor online LLMs (zoals OpenAI's GPT-4/3.5 en Anthropic's Claude). Er is beperkte ondersteuning om *lokale modellen* te gebruiken, maar dit vereist enige configuratie en heeft nuances. Zo werkt Cursor offline niet out-of-the-box; zelfs als je een lokaal LLM aansluit, gebruikt Cursor nog steeds zijn cloud-API's voor bepaalde taken zoals indexeren van de codebase . In de praktijk kun je via tools (bijv. via het Model Context Protocol of integraties als Ollama) een lokaal model laten draaien voor de AI-respons, maar Cursor blijft internet nodig hebben voor context management tenzij je ook het indexeren/embedding-proces vervangt . Een lokaal model zal vaak een kleinere context window hebben (veel open-source modellen ondersteunen bv. 4k of 8k tokens), wat betekent dat Cursor nog agressiever moet samenvatten of context selecteren. In dergelijke gevallen kan de workflow er uitzien als: Cursor bepaalt relevante context (bijv. via vector search of heuristiek) en geeft dat plus je prompt aan het lokale model, en ontvangt van dat model de output. De *rol van een lokaal model* is hier simpelweg die van vervanger voor de cloud-LLM voor het genereren van code of antwoorden, maar **het contextmanagement zelf blijft grotendeels door Cursor uitgevoerd**. Het lokale model speelt dus geen magische extra rol in het onthouden van regels – het volgt gewoon de prompt die Cursor opstelt. Samengevat: lokaal modellen **kunnen** worden geïntegreerd, maar je bent vaak beperkt tot online indexing of moet zelf een alternatief opzetten; volledig offline werken is (nog) niet standaard mogelijk in Cursor's workflow .

Regels in Cursor: van globale settings tot projectregels

Globale regels (Rules for AI):

Cursor biedt in de instellingen een veld *"Rules for AI"* waar je globale instructies aan de AI kunt meegeven . Dit zijn regels of richtlijnen die voor **alle** projecten en sessies gelden. Denk

aan code style voorkeuren, taalinstellingen, of veiligheidsgordels voor de AI. Regels die je hier instelt worden door Cursor als vaste component in elke prompt opgenomen (meestal als een systeemprompt bovenaan). Ze zijn als het ware *universeel*: ongeacht welk project je opent, deze richtlijnen zijn van kracht . Volgens de documentatie worden deze globale regels in alle projecten toegepast zodra je ze in Cursor's instellingen opslaat . Het is verstandig om hier alleen algemene zaken in te zetten (bijv. "Schrijf altijd duidelijke comments" of "Geef de voorkeur aan functionele programmeerstijl") en project-specifieke details te laten voor de projectregels.

Project-specifieke regels (.cursorrules):

Voor richtlijnen die alleen voor een bepaald project gelden heeft Cursor de mogelijkheid van een .cursorrules bestand in de root van je project. Dit was de oorspronkelijke methode om projectregels te definiëren. In dat tekstbestand (meestal Markdown of JSON formaat) kun je beschrijven hoe de AI zich moet gedragen specifiek voor *jouw* project – bv. architectuurprincipes, file structure uitleg, naming conventions, welke libraries te gebruiken of te vermijden, etc. Wanneer je Cursor opent in dat project, zal het proberen deze .cursorrules inhoud toe te voegen aan de context voor chat- of code-opdrachten . Je kunt dit zien als een *projectbrede systeemprompt*. In tegenstelling tot de globale regels (die overal gelden), worden deze alleen toegepast voor dat specifieke project waar het bestand aanwezig is.

Fijnmazige regels per onderdeel (.cursor/rules/):

Recente versies van Cursor (v0.45 en hoger) introduceren een verbeterde aanpak: in plaats van één monolithisch .cursorrules bestand, kun je nu een directory .cursor/rules/ aanmaken met meerdere regelbestanden. Dit maakt *contextuele regels* mogelijk die gelden voor specifieke onderdelen van je project. Zo zou je aparte regel-bestanden kunnen hebben voor frontend vs. backend code, of voor unit tests vs. documentatie. Volgens de Cursor community kun je hiermee meerdere repository-niveau regels op schijf zetten, waarbij **de AI-agent automatisch kiest welke regel(s) relevant zijn** voor de huidige context . Je kunt bijvoorbeeld in .cursor/rules/ een frontend.md en backend.md plaatsen, en Cursor zal op basis van het pad of bestandsnaam bepalen welke instructies in te laden (je kunt per regelbestand ook glob-patronen of paden aangeven waarop het van toepassing is) . Dit biedt *granulaire controle*: als je een testbestand bewerkt, zal Cursor bijvoorbeeld de regels uit tests.md laden, maar niet onnodig de regels voor productieve code. Het scheiden van regels maakt ze makkelijker te onderhouden én beperkt de tokens die regels innemen tot alleen wat nodig is. Intern behandelt Cursor deze map als verzameling van mogelijke systeemprompts en het kiest de meest relevante (of combineert er enkele) afhankelijk van waar je mee bezig bent .

Hoe worden regels verwerkt in de workflow?

Wanneer je een nieuwe chat start of de AI in de code-editor aanroept, kijkt Cursor eerst of er relevante regels zijn en zet die bovenaan de context (voor de gebruikersprompt). Meestal zal de volgorde in de promptcontructie zoiets zijn als: *(1) globale regels*, *(2) projectregels uit*

.cursorrules of .cursor/rules, *(3) eventueel een automatische beschrijving van de huidige file als Cursor die genereert*, gevolgd door *(4) de door de gebruiker gevraagde prompt of instructie*. De regels fungeren dus als *systeembericht* waarin de AI verteld wordt hoe hij zich moet gedragen voordat hij jouw daadwerkelijke vraag/code-opdracht ziet.

In Cursor's instellingen kun je ook direct meerdere regels aanmaken die dan in .cursor/rules verschijnen. Men kan per regel in die UI ook instellen op welke bestanden of directories deze van toepassing is (via path of extensie filters) . Dit stroomlijnt het proces: je hoeft niet handmatig de JSON/Markdown te schrijven; Cursor genereert de structuur. Intern is het effect hetzelfde: de relevante regeltekst wordt aan de prompt toegevoegd zodra je met de bijpassende context werkt.

Modelverschillen in regels toepassen:

Een opmerkelijk punt is dat de effectiviteit van het regelsysteem soms per model verschilt. Zo merkte een gebruiker op dat zijn personality-regels (in een .cursor/rules/personality.md bestand) wel werden toegepast met Claude 3.5 Sonnet, maar niet met andere modellen . Met Claude kreeg hij dus automatisch de gewenste "persoonlijkheid" in elke chat of composer-venster, terwijl GPT-4 er blijkbaar langsheen keek. Dit suggereert dat Cursor's regelinjectie mogelijk (destijds) specifiek beter werkte of alleen ondersteund werd voor bepaalde modelintegraties. Het kan ook zijn dat Claude, door zijn grotere context of door hoe hij de prompt leest, de regels beter "zag" dan GPT. In het algemeen zou Cursor de regels model-agnostisch moeten meesturen, maar deze ervaring laat zien dat in de praktijk de naleving per model kon verschillen. Claude 3.5 Sonnet staat er overigens om bekend dat het consistent de aangeleverde instructies volgt , wat kan verklaren dat regels daar merkbaar effect hebben. GPT-modellen hebben een harde scheiding tussen *system* en *user* messages; Cursor moet de regels dus als system message meesturen. Als dat niet correct gebeurde of werd overschreven door iets, kan GPT de regels hebben gemist. Dit is iets om op te letten: controleer bij wisseling van model of je regels nog worden gerespecteerd.

Workflow: integratie van AI-modellen en de rol van Claude 3.5 "Sonnet"

Keuze van modellen in Cursor:

Cursor ondersteunt meerdere AI-modellen. In de gratis versie zijn dat vaak open modellen (bijv. Code Llama via OpenRouter) of je eigen API-sleutels voor OpenAI. De Pro-versie van Cursor biedt direct integratie met krachtige modellen als Anthropic's Claude 1.3/1.4 ("Claude Instant" varianten) die binnen Cursor de marketingnaam *Claude 3.5 - Sonnet* hebben gekregen. Veel gebruikers geven aan dat *Claude 3.5 Sonnet* hun voorkeur heeft als LLM in Cursor vanwege de combinatie van grotere context en consistentie in het opvolgen van instructies . Claude is getraind op uitgebreide codegerelateerde datasets en heeft een context window tot wel 100k tokens, wat bij omvangrijke projecten een enorm voordeel biedt.

Integratie van Claude 3.5 Sonnet:

Om Claude Sonnet te gebruiken moet je doorgaans een Anthropic API key opgeven in Cursor (of via een service als OpenRouter verbinden). Cursor stuurt dan je prompts naar Anthropic's API in plaats van bijvoorbeeld OpenAI. Het *workflow* verschil merk je vooral in hoeveel context je kunt meesturen en hoe de antwoorden geformuleerd zijn. Zoals eerder genoemd beperkt Cursor standaard de context tot ~10k tokens zelfs als een model meer aankan . Echter, als je een Long-Context Chat opent of in de instellingen aangeeft die grotere context te willen benutten, kan Claude de volledige 100k aan relevante projectinhoud ontvangen. Dit betekent dat Cursor bijvoorbeeld hele documenten of veel meer files tegelijk kan meesturen zonder te hoeven samenvatten, zolang het binnen Claude's limiet blijft. De integratie zorgt er dus voor dat de *bottleneck* van context eerder in Cursor's software (10k default) zat dan in Claude zelf.

Workflow in de praktijk (Claude als voorbeeld): Stel je begint een chat in Cursor met Claude Sonnet als model. Cursor kijkt naar je huidige project, neemt de global rules en project rules, voegt misschien een automatische beschrijving toe (sommige versies genereren een korte samenvatting van de geopende file of project) en plaatst al deze instructies in het prompt. Dan komt jouw eigen vraag of opdracht. Dit geheel gaat naar Claude's API. Claude verwerkt het en dankzij zijn consistente opvolgen van systeeminstructies zal hij bijvoorbeeld netjes jouw coding style hanteren en alleen de gevraagde output geven (ervan uitgaande dat je regels dat specificeren). Gebruikers hebben gemerkt dat Claude soms zelfs iets te gehoorzaam is: hij kondigt bijvoorbeeld aan wat hij gaat doen voordat hij de code toont, ondanks dat ze vroegen dit niet te doen . Dit is een model-quirk van Claude 1.x en kan met regels deels worden afgeleerd. De output komt terug en wordt in de Cursor UI getoond, eventueel inline in de code if you're in composer mode.

Lokale modellen in de workflow:

Mocht je een lokaal model geïntegreerd hebben (bijv. via Ollama met een CodeLlama variant), dan verloopt de workflow vergelijkbaar, maar in plaats van een API call naar een cloudmodel, stuurt Cursor de prompt naar de lokale service op je machine. Omdat lokale modellen vaak minder tokens aankunnen en minder krachtig zijn, zul je merken dat je vaker context moet beperken. Wel kun je daardoor geheel offline coderen zodra de context klaarstaat (mits Cursor niet stiekem iets online blijft doen). Zoals eerder genoemd vergt offline mode normaliter toch internet voor dingen als embeddings ophalen, tenzij je daar een oplossing voor hebt . Sommige community-projecten combineren Cursor met externe tools (zoals *Pieces for Developers* of *Weaviate*) om lokaal een vector database te draaien van de code. Dan kan Cursor via MCP relevante code opvragen zonder alles in de prompt te hoeven houden, wat een soort "oneindig contextvenster" benadering geeft. Een ontwikkelaar toonde bijvoorbeeld hoe integratie met PiecesOS een *unlimited context window* mogelijk maakt: je kunt zoveel files "uploaden" als je wilt en de AI haalt op verzoek relevante stukken eruit, in plaats van op token-limiet alles mee te sturen . Dit zijn echter meer experimentele uitbreidingen op de standaard Cursor workflow.

Samenvatting: Over het geheel genomen is de Cursor workflow zo ontworpen dat **de AI-model laag (Claude, GPT, lokaal, etc.) wordt gevoed met een zorgvuldig samengestelde prompt** die zowel jouw instructie als de nodige context (code + regels) bevat. Cursor zelf beheert wat erin komt, gebruikmakend van de regels die je instelt en de content van je project. Het model genereert daarop een antwoord of code wijziging, en Cursor dient dat weer aan jou terug in de IDE. Afhankelijk van het gekozen model zie je verschillen in maximale context, responsstijl en betrouwbaarheid in opvolgen van regels.

Verdwijnen van regels uit de context (FIFO-probleem)

Een bekende uitdaging bij lange AI-sessies is dat eerdere *system prompts* of instructies na verloop van tijd buiten de context window kunnen vallen. Dit geldt ook voor Cursor. **Ja, regels kunnen na verloop van tijd uit de context verdwijnen** als het gesprek of de interactie langdurig is en er veel nieuwe inhoud bij komt. Omdat Cursor (buiten long-context mode) een sliding window van ~10k tokens hanteert, zullen aanvankelijk meegegeven regels naar de achtergrond verdwijnen zodra de conversatie dat limiet overschrijdt. Dit is vergelijkbaar met hoe chatGPT oudere berichten “vergeet” als het gesprek heel lang wordt.

Gevolgen in Cursor:

Stel je hebt in .cursorrules staan dat de AI bij voorkeur alleen de *aangepaste code* moet geven en niet steeds de ongewijzigde originele code erbij (om diffs compact te houden). In het begin zal Cursor die regel meesturen en zal het model zich eraan houden. Maar na een flink aantal interacties, als de context volloopt, loop je het risico dat deze instructie niet meer in de laatste 10k tokens zit die verstuurd worden. Het resultaat: de AI begint toch weer beide versies van de code te geven, omdat het “vergeten” is dat het dat niet mocht. Er is daadwerkelijk een forumklacht hierover: een gebruiker had expliciet in Rules for AI gezet om alleen de aangepaste code te geven, *toch* bleef het model ook de originele code tonen . Dit klinkt als een typisch geval van een regel die niet consequent werd toegepast in latere prompts.

Cursor zelf biedt (vooralsnog) geen magische permanente geheugenopslag voor regels – het is aan de promptconstructie. In vroege versies was er zelfs een bug/misfeature dat .cursorrules niet bij **elke** prompt werd meegestuurd. Een gebruiker meldde dat de projectregels niet betrouwbaar bij elke vraag werden meegestuurd, met name bij de code-compositie (*Cursor* mode) . Dit soort inconsistentie zorgt ervoor dat een regel soms wegvalt. De ontwikkelaars adviseerden toen als tijdelijke oplossing om *handmatig* die regels in je vraag of context te plakken, of een speciaal *marker* toe te voegen om Cursor te dwingen ze erbij te pakken . Dit bevestigt dat regels niet automatisch persistent blijven als de context het niet toelaat.

Het FIFO-probleem is dus reëel: **oude instructies verdwijnen als er nieuwe bijkomen en de limiet is bereikt**. Zonder ingreep zullen je zorgvuldig gedefinieerde regels na verloop van tijd hun effect verliezen. Voor de gebruiker manifesteert dit zich als het model dat ineens ander stijl of ongewenste output geeft, die je aanvankelijk via regels had proberen te voorkomen.

Regels persistent houden: best practices en oplossingen

Gezien bovenstaande, hoe kunnen we zorgen dat regels consistent toegepast blijven? Er zijn een aantal *best practices* en mogelijke oplossingen om instructies persistent in de context te houden:

• **Hou regels bondig en doelgericht:** Hoe minder tokens je regels innemen, hoe langer ze in een beperkt contextvenster mee kunnen. Probeer dus je .cursorrules of regels in de UI niet te overladen met essay-achtige teksten. Focus op de belangrijkste punten (coding style, conventies, do's/don'ts). Korte, krachtige regels passen makkelijker in elk prompt zonder de context te domineren.

• **Gebruik gestructureerd formaat voor regels:** Ervaring uit de community leert dat het format van je regels uitmaakt voor hoe goed het model ze onthoudt. Zo wordt aangeraden om regels in een gestructureerde opmaak zoals JSON te schrijven in plaats van vrije tekst of Markdown. Dit maakt het voor de AI duidelijker dat het om formele instructies gaat. Eén ontwikkelaar stelt zelfs dat *"5 minuten op het Cursor forum je zullen vertellen dat JSON 100x beter werkt dan Markdown voor cursorrules"* . Bijvoorbeeld, in plaats van een lang proza in Markdown, kun je een JSON-object maken met velden als "coding_style": "... beschrijving ...", "target_framework": "...", etc. Dit dwingt het model om de info als data te zien. Veel gebruikers melden dat de AI consistenter is zodra ze dit deden .

• **Projectregels opsplitsen (.cursor/rules):** Maak gebruik van de mogelijkheid om je regels op te delen per context. Zoals eerder besproken zorgt een map .cursor/rules met specifieke deelregels ervoor dat **niet alle regels altijd mee hoeven**, maar alleen de relevante. Dit voorkomt dat je onnodig tokens verspilt aan bijvoorbeeld database-regels terwijl je in de frontend code aan het werk bent. Bovendien zijn de deelbestanden meestal korter individueel, waardoor ze makkelijker in de context passen. Cursor's agent selecteert automatisch de juiste regel op basis van de situatie , maar controleer of je de globs/patronen goed hebt ingesteld. Zo blijven de juiste regels *telkens aanwezig* precies waar je ze nodig hebt.

• **Herhaal belangrijke regels indien nodig:** Als je merkt dat een bepaalde richtlijn cruciaal is en de AI begint af te wijken na lange sessies, schroom niet om die regel opnieuw te benadrukken. Je kunt dit handmatig doen ("Even ter herinnering: volg onze stijlguide die zegt X en Y."), of – als je zelf aan de tooling sleutelt – door programmatically de systeemprompt opnieuw aan te vullen. Bijvoorbeeld, sommige ontwikkelaars injecteren periodiek (om de paar interacties) de belangrijkste regels opnieuw bovenaan de context. Dit is een brute-force maar effectieve manier om FIFO-verlies tegen te gaan: je *verlengt de levensduur* van de regels door ze telkens weer als 'nieuw' erin te zetten. Een Cursor medewerker suggereerde ook dat je een *specifieke marker* in je prompt kunt plaatsen die Cursor triggert om de regels toe te voegen . (Hoewel niet gedetailleerd publiek gedocumenteerd, zou zo'n marker bijvoorbeeld een

speciale comment of token kunnen zijn die Cursor herkent – het idee is dat je in je vraag iets zet als [INCLUDE_RULES] om zo de regels weer op te nemen zonder ze volledig uit te schrijven.)

• **Context resetten of opsplitsen:** In plaats van oneindig door te vragen in één sessie, kan het handig zijn om het gesprek op te knippen. Als je merkt dat de context vol raakt (bij 10k tokens is dat al een hoofdstuk in een boek), overweeg dan **een nieuwe chat te starten** binnen Cursor voor een volgend deel van de taak. In die nieuwe chat begin je weer met een schone lei waar de .cursorrules en globale regels weer fris worden ingeladen als basis. Je kunt eventueel een samenvatting van het voorgaande meegeven, maar de regels staan gegarandeerd weer bovenaan. Op deze manier voorkom je dat instructies verdampen naarmate je verder gaat; je *herstart het geheugen* met behoud van regels. Natuurlijk vereist dit wat handmatige coördinatie (en mogelijk contextverlies van het directe gesprek), dus het is een afweging tussen consistentie vs. continuïteit.

• **Gebruik long-context mode voor cruciale regels:** Als je toegang hebt tot een model met een zeer grote context (zoals Claude 100k tokens), maak er dan gebruik van via Cursor's long-context functie. Hiermee vergroot je de kans dat zelfs lange .cursorrules of meerdere regelbestanden nog lang in het contextvenster blijven staan voordat ze eruit geduwd worden. Het model zelf kan dan immers veel meer geschiedenis *onthouden*. Let wel dat Cursor zelf bij normaal gebruik toch tot ~10k beperkt, dus expliciet Long-Context Chat gebruiken is nodig . Dit is vooral nuttig voor marathon-sessies waar je echt alles wilt blijven meenemen.

• **Retrieval-gerichte oplossingen:** Een meer geavanceerde techniek is om geen volledige afhankelijkheid te hebben van het conversationele geheugen, maar context op te slaan buiten het model en op te halen wanneer nodig. Dit principe staat bekend als *Retrieval Augmented Generation (RAG)*. Concreet zou je bijvoorbeeld je .cursorrules ook kunnen opnemen in een vector-database of sleutel-waarde store. Telkens voor je een prompt stuurt, check je via een sleutelwoord of embedding of de regels relevant zijn en voeg je ze (of een relevante subset) toe. Zo'n systeem kan in theorie oneindig veel "kennis" hebben en alleen de relevante relevant maken. In de context van Cursor doen integraties als **PiecesOS** iets vergelijkbaars: ze bewaren grote hoeveelheden projectinformatie extern en Cursor vraagt op commando om bijv. *"Geef alle relevante info over module X"* die dan uit die store komt en in de prompt gevoegd wordt . Hoewel dit niet standaard in Cursor zit, is het wel een techniek die helpt om de schaarse contextruimte in het LLM te sparen voor echt essentieel materiaal. Voor regels zou dit betekenen dat je niet per se alle regels constant meestuurt, maar ze bijvoorbeeld tagged in een DB hebt en dat een query naar "style rules" ze ophaalt indien nodig.

• **Tooling en toekomstige features:** Realiseer je dat Cursor zelf continu in ontwikkeling is. De community heeft al feature requests ingediend om het omgaan met .cursorrules te verbeteren. Zo werd voorgesteld dat Cursor automatisch deze regels moet handhaven vergelijkbaar met hoe een .gitignore altijd gerespecteerd wordt – met andere woorden, dat de tool er zelf voor zorgt dat jouw projectregels nooit vergeten worden. Wees dus alert op updates van Cursor: nieuwe versies zouden dit proces kunnen automatiseren, bijvoorbeeld door regels altijd als *pinned system message* mee te sturen ongeacht de geschiedenis. Totdat zulke features komen, blijf zelf proactief in het bewaken van je AI's output op naleving van je richtlijnen.

Aanbevelingen in het kort:

• Definieer een beknopte set van globale regels voor algemene voorkeuren.

• Gebruik projectregels (.cursorrules of nog beter .cursor/rules/) voor project-specifieke guidance; splits ze op per domein om tokenruimte te sparen.

• Formatteer regels in een duidelijk schema (JSON/YAML) om interpretatie te verbeteren .

• Houd de contextgrootte in de gaten; bij lange sessies herhaal of verfris regels handmatig indien nodig.

• Profiteer van grotere context-modellen (Claude Sonnet, GPT-4 32k) wanneer beschikbaar, om minder snel in geheugenlimieten te lopen .

• Experimenteer met geavanceerde oplossingen (zoals externe knowledge stores of markers) als jouw use-case extreem langdurige context vereist.

• Blijf betrokken bij de Cursor community/forums voor tips – gebruikers delen daar geregeld *templates* en voorbeelden van goed werkende .cursorrules configuraties voor allerlei frameworks, wat je inzicht kan geven in hoe je effectief regels formuleert en toepast.

Conclusie

Cursor's contextmanagement is krachtig maar kent grenzen. Standaard krijg je ~10k tokens aan geheugen, waardoor je soms moet kiezen welke informatie *altijd* vooraan moet staan – jouw regels horen daar in principe bij. Cursor faciliteert dit via globale- en projectregels, en heeft met de .cursor/rules directory een stap gezet naar slimmere contextafhankelijke instructies. AI-modellen zoals Claude 3.5 Sonnet integreren naadloos met Cursor en bieden extra grote context windows en consistent gedrag, wat helpt om minder vaak tegen de contextlimiet aan te lopen. Toch blijft het bewaken van de *persistentie van regels* belangrijk: als een sessie erg lang wordt, kunnen instructies van het begin wegvallen door het FIFO-effect.

Gelukkig zijn er strategieën om hiermee om te gaan: van het frequent herinjecteren van regels, tot het gebruik van long-context modes of het opsplitsen van gesprekken. Best practices zoals het kort en gestructureerd houden van regels en het context-specifiek laden ervan vergroten de kans dat de AI ze volgt van begin tot eind. Ook kun je inspiratie putten uit vergelijkbare technieken in de bredere LLM-wereld, zoals retrieval systemen en samenvattingsmethoden, om jouw AI-assistent "bij de les" te houden.

Kortom, **Cursor gaat slim om met zijn context window en regels, maar het is aan de gebruiker om de AI scherp te houden**. Met de juiste inrichting van regels en wat oplettendheid kun je Cursor's AI consistent laten handelen naar jouw wensen, zelfs in langere ontwikkelingssprints. En met toekomstige verbeteringen – mogelijk automatische persistentie van regels – zal dit alleen maar beter worden. Neem deze inzichten mee in je eigen workflow om het maximale uit Cursor en zijn AI-modellen te halen.