Les 13 — Lesstof

Agents — een Polderfest-app die zelf nadenkt

Vak:	AI-Assisted Development
Klas:	Klas A (3 uur fysiek)
Vervolg op:	Les 12 — Tool Calling
Volgende les:	Les 14 — Externe APIs + Cursor + Vercel deploy

Vak: AI-Assisted Development **Opleiding:** NOVI Hogeschool Utrecht **Vorige les:** Les 12 — Tool Calling **Volgende les:** Les 14 — Externe APIs + Cursor + Vercel deploy

Inhoud

1. Wat is een agent?

Een **agent** is een Large Language Model dat in een loop draait, tools gebruikt, en zelfstandig kiest welke volgende stap nodig is om een doel te bereiken.

Een voorbeeld maakt het concreet. Stel een gebruiker stelt deze vraag aan onze Polderfest-app:

> "Plan een zaterdagavond met indie en techno, geen tijdoverlap, alleen onder een dak als het regent."

Hierin zitten meerdere vragen verstopt:

- Welke indie-bands spelen er zaterdag?

- Welke techno-bands spelen er zaterdag?
- Wat is het schema (om overlap te checken)?
- Hoe is het weer op zaterdag?
- Kunnen we al die info combineren tot één avondprogramma?

Een gewone chatbot beantwoordt één vraag tegelijk. Een agent pakt alle vijf de sub-vragen autonoom op, in de juiste volgorde, en levert een eindantwoord.

2. Tool calling vs Agent — een continuüm

Belangrijk om vooraf te zeggen: **er is geen harde lijn tussen tool calling en agents.** In de Vercel AI SDK v6 gebruik je voor beide dezelfde syntax: `streamText`, `tool()`, `stopWhen`.

Het verschil zit in **hoe je het ontwerpt en gebruikt**.

Aspect	Tool calling (Les 12)	Agent (Les 13)
Vraag-type	"Voer deze functie uit"	"Bereik dit doel — kies zelf hoe"
Aantal stappen	Meestal 1-3 (ook met stopWhen)	Vaak 4-10+, variabel
Wie kiest de volgorde	Jouw prompt stuurt sterk	AI plant zelf, autonomer
Bronnen combineren	Per tool één bron	DB + externe API + ... door elkaar
Reflectie / bijsturen	Nee — direct antwoord	AI checkt resultaat, beslist vervolg

Belangrijk: tool calling kón in Les 12 al externe APIs aanroepen. De `execute()` van een tool is gewoon JavaScript — een `fetch` is prima. Het is niet alsof tool calling daar beperkt in is.

Wat agents onderscheidt is dat ze **meerdere bronnen door elkaar combineren** in één antwoord, omdat ze meer stappen mogen zetten en zelf de volgorde bepalen.

3. Wanneer voegt een agent écht waarde toe?

Tool calling is genoeg als:

- Je weet welke functie nodig is voor de vraag (een DB-zoekopdracht, een API-call)
- Het antwoord ligt in één bron
- Het pad is voorspelbaar — je kunt in je eigen prompt al beschrijven "doe eerst X, dan Y"

Een agent is écht zinvol als:

- De vraag is open: "plan", "vergelijk", "vind het beste", "concludeer"
- De AI moet meerdere bronnen combineren in één antwoord
- Het pad is onvoorspelbaar — soms drie stappen, soms zeven
- Het eerste resultaat triggert vervolgvragen ("die band is uitverkocht — zoek alternatief")

Waarom niet altijd een agent?

Agents lopen langer. Meer stappen = meer tokens = hogere kosten en hogere latency. Ook lastiger te debuggen door non-determinisme — twee dezelfde vragen kunnen verschillende paden bewandelen.

Pragmatisch advies: start altijd met tool calling. Schaal pas naar agent-niveau als je merkt dat je system prompt steeds langer wordt om "doe eerst dit, dan dat" voor te schrijven — dan kan de AI dat beter zelf bepalen.

4. De drie AI SDK functies die we gebruiken

4.1 `tool({ description, inputSchema, execute })`

Dezelfde functie als in Les 12. Drie velden:

```
import { tool } from "ai";
import { z } from "zod";

searchBands: tool({
  description: "Zoek bands op genre en/of dag.",
  inputSchema: z.object({
    genre: z.string().optional(),
    day: z.enum(["vrijdag", "zaterdag", "zondag"]).optional(),
  }),
  execute: async ({ genre, day }) => {
    // Supabase query
    return data;
  },
}),
```

- `description` — wat de AI hierover leest om te beslissen of-ie de tool wil gebruiken
- `inputSchema` — Zod-schema, AI vult de parameters automatisch in
- `execute` — wat draait als de tool wordt aangeroepen. Gewoon JavaScript: DB-query, fetch, computatie

4.2 `stopWhen` — de "knop" om de loop te bouwen

```
import { stepCountIs } from "ai";

streamText({
  model: openai("gpt-5.2"),
  messages,
  tools: { ... },
  stopWhen: stepCountIs(8),
});
```

`stopWhen` is een conditie waarop de loop stopt. `stepCountIs(N)` is de meest gebruikte: stop na maximaal N stappen.

In Les 12 hebben we dit ook gebruikt, vaak met `stepCountIs(3)`. Voor agents zetten we 'm typisch hoger — 8 of meer — omdat we de AI ruimte willen geven om zelf het juiste pad te kiezen.

Andere stop-condities (combineerbaar):

```
stopWhen: [
  stepCountIs(8),
  hasToolCall("finishPlan"),  // stop als specifieke tool is aangeroepen
],
```

`hasToolCall` is handig als je de AI dwingt een "klaar"-tool aan te roepen voor-ie stopt.

4.3 `hasToolCall` — stop op specifieke tool (NIEUW)

Combineer `stepCountIs` met een **stop-zodra-tool-aangeroepen** conditie:

```
import { stepCountIs, hasToolCall } from "ai";

stopWhen: [stepCountIs(8), hasToolCall("finishPlan")],
```

Effect: zodra de agent een tool genaamd `finishPlan` heeft aangeroepen, stopt de loop direct — ongeacht hoeveel stappen er nog over zijn. Dit forceert een nette afsluiting.

Combineer dit met een tool die **structured output** vraagt:

```
finishPlan: tool({
  description: "Roep dit ALS LAATSTE aan met een gestructureerd plan.",
  inputSchema: z.object({
    summary: z.string(),
    items: z.array(z.object({
      time: z.string(),
      band: z.string(),
      stage: z.string(),
      reason: z.string(),
    })),
  }),
  execute: async ({ summary, items }) => ({ summary, items, ok: true }),
}),
```

De UI kan dit object renderen als een mooie timeline of kaartenset. Geen risico op vrije-vorm-tekst die je moet parsen.

4.4 `prepareStep` — dynamisch model per stap (NIEUW)

Geef per stap ander gedrag:

```
streamText({
  model: openai("gpt-5.2"),   // default

  prepareStep: ({ stepNumber }) => {
    if (stepNumber === 0) {
      return { model: openai("gpt-5.2") };     // krachtig — planning
    }
    return { model: openai("gpt-5-mini") };     // goedkoper — uitvoering
  },

  // ...tools, stopWhen, etc.
});
```

Eerste stap (begrip + planning) = duur, krachtig model. Latere stappen (simpele tool-calls uitvoeren) = goedkoop, snel model. Zelfde resultaat, significant lagere kosten en latency.

`prepareStep` kan ook andere dingen aanpassen per stap: `activeTools` (welke tools beschikbaar zijn), `system` prompt, etc. Voor vandaag gebruiken we het alleen voor model-switching.

5. Externe APIs als tool

Tools hoeven niet alleen je eigen Supabase-database te zijn. Externe APIs werken net zo. Een tool is gewoon een functie — wat erin gebeurt is aan jou.

Voor onze Polderfest-agent gebruiken we **Open-Meteo** — een gratis weer-API die geen account of API key nodig heeft. Perfect voor demos en onderwijs.

```
getWeather: tool({
  description: "Weer-forecast voor een festivaldag.",
  inputSchema: z.object({ date: z.string() }),
  execute: async ({ date }) => {
    const url = new URL("https://api.open-meteo.com/v1/forecast");
    url.searchParams.set("latitude", "52.0907");
    url.searchParams.set("longitude", "5.1214");
    url.searchParams.set("daily", "temperature_2m_max,precipitation_probability_max");
    url.searchParams.set("start_date", date);
    url.searchParams.set("end_date", date);
    url.searchParams.set("timezone", "Europe/Amsterdam");
    const res = await fetch(url);
    const data = await res.json();
    return {
      temperatureMaxC: data.daily.temperature_2m_max[0],
      rainChancePercent: data.daily.precipitation_probability_max[0],
    };
  },
}),
```

Vuistregel: wat een fetch kan, kan een tool. Stripe-saldo opvragen. GitHub-issues lezen. Wikipedia. Spotify. Alles via HTTP — alles kan een tool zijn.

Dit is een mogelijkheid die ook in tool calling van Les 12 al bestond. Het verschil bij agents: omdat de loop meerdere stappen mag duren, kan de agent het *resultaat* van een externe API gebruiken om vervolgens een tweede tool aan te roepen op basis daarvan. Dat is de échte meerwaarde.

5.1 Write-tools — de wereld veranderen, niet alleen lezen

Tot nu toe lazen al onze tools data uit Supabase of externe APIs. Tools kunnen óók schrijven. Dat opent een hele nieuwe categorie: agents die **iets doen** ipv. alleen iets opzoeken.

```
addToFavorites: tool({
  description: "Voeg een band toe aan de favorieten van de gebruiker.",
  inputSchema: z.object({
    bandName: z.string(),
  }),
  execute: async ({ bandName }) => {
    // 1. Zoek band op naam (we hebben band_id nodig — FK in user_favorites)
    const { data: band } = await supabase
      .from("bands").select("id, name")
      .ilike("name", `%${bandName}%`).limit(1).single();
    if (!band) return { error: `Band niet gevonden` };

    // 2. Insert in bestaande user_favorites tabel
    const { data, error } = await supabase
      .from("user_favorites")
      .insert({ user_email: "demo@polderwave.app", band_id: band.id })
      .select().single();
    if (error) return { error: error.message };
    return { added: true, band: band.name, favoriteId: data.id };
  },
}),
```

De gebruiker zegt *"Zet Band 5 in mijn favorieten."* — de agent roept `addToFavorites` aan — er staat een rij in `user_favorites` met `band_id` als foreign-key naar `bands`. We gebruiken de bestaande tabel uit Les 11/12, geen nieuwe tabel nodig. Voor demos hardcoderen we `user_email` op `"demo@polderwave.app"`; in productie gebruik je echte auth + RLS (zie Les 10).

Belangrijke veiligheidsoverweging: write-tools zijn machtig én riskant. Een agent die kan schrijven kan ook fout schrijven. In productie:

- Beperk wat een tool mag (alleen INSERT op specifieke tabellen)
- Beperk per user (Supabase RLS — Les 10)
- Log alle writes voor auditability

6. De agent-loop

Visueel:

Soms duurt het twee stappen. Soms zes. Soms acht — dan kicked `stepCountIs(8)` in. De agent beslist.

7. Stappenplan: Polderfest → Polderfest-met-agent

Compleet stap-voor-stap met code-blokken: zie `lesbestanden/Les13-Stap-voor-stap.md`.

Samenvatting van de 11 stappen:

1. Polderfest-original uitpakken (Les 12 versie) of eigen versie gebruiken
2. Zod check — al geïnstalleerd
3. Maak nieuwe route `app/api/agent/route.ts`
4. Voeg eerste tool toe: `searchBands` (Supabase)
5. Voeg `stopWhen: stepCountIs(8)` toe
6. Voeg tweede tool toe: `getStageSchedule`
7. Voeg derde tool toe: `getWeather` (externe API, Open-Meteo)
8. Schrijf system prompt
9. Maak UI-pagina `app/agent/page.tsx`
10. Testen in browser met drie voorbeeld-prompts
11. Deploy naar Vercel

Twee referentie-zips staan in `lesbestanden/`:

- `polderfest-original.zip` — startpunt
- `polderfest-with-agent.zip` — referentie als je vastloopt

8. Veelvoorkomende fouten en debug-tips

Probleem	Oplossing
Agent stopt na 1 tool-call	Heb je stopWhen: stepCountIs(8) toegevoegd?
Tool wordt nooit aangeroepen	Description is te vaag — schrijf in heldere taal wat de tool doet
Tool wordt te vaak aangeroepen	System prompt is te open — wees specifieker
Loop blijft hangen / oneindig	Check stopWhen — zonder maximum kan hij hard doorgaan
Open-Meteo geeft niets terug	Check ISO datum-format: YYYY-MM-DD
UI toont geen tool-calls	Loop door m.parts heen, herken p.type.startsWith("tool-")
Supabase error in agent-route	Check SUPABASE_SERVICE_ROLE_KEY in .env.local
Vercel deploy faalt	Env vars in Vercel toegevoegd? Build logs lezen

Debug-tip: zet `console.log` in elke `execute` om te zien welke tools de agent aanroept en in welke volgorde. Heel leerzaam om dit één keer te doen.

Verder lezen

- Vercel AI SDK docs over tools en agents:
- Anthropic over "Workflows vs Agents":
- Open-Meteo API: