Spec-Kit Live-Demo · Manuskript

Demo-Projekt: **`askclaude`** — eine minimale Next.js-App, die die Streaming-API des `@anthropic-ai/sdk` direkt im Browser zeigt. Eingabe → „Ask" → Antwort tröpfelt Token für Token rein.

Klein genug für 5–7 Min live, groß genug, um Constitution / Spec / Plan / Tasks zu rechtfertigen. Bonus: thematisch passend — wir bauen *mit* Spec-Kit eine App, die das SDK aus Phase 2 in Aktion zeigt.

0 — Setup (vorab, off-stage)

```
uv tool install specify-cli --from git+https://github.com/github/spec-kit.git@v0.0.55
mkdir askclaude && cd askclaude && git init
specify init --here --integration claude
echo "ANTHROPIC_API_KEY=sk-ant-..." > .env.local
```

Copy

Dann `claude` im Verzeichnis starten. Ab hier passiert alles im Agenten.

1 — `/speckit.constitution`

> Briefing-Text zum Einkleben:

```
Erstelle eine Constitution für eine Next.js-Demo-App namens "askclaude".

Prinzipien:
- Next.js 15, App Router, Server Components by default,
  Client Components nur für Interaktivität.
- TypeScript strict, kein implicit any.
- Anthropic-SDK direkt einbinden (@anthropic-ai/sdk).
  Keine Wrapper, kein LangChain, kein Vercel-AI-SDK.
- API-Key kommt aus .env.local (ANTHROPIC_API_KEY) und bleibt
  ausschließlich serverseitig — niemals an den Client geben.
- Streaming-Antworten via ReadableStream, niemals JSON-Polling.
- Runtime nodejs (nicht edge), damit der Anthropic-SDK-Default trägt.
- Styling minimal: Tailwind v4, das create-next-app mitbringt.
- Keine DB, kein Auth, keine Persistenz, keine Telemetrie — Demo-Scope.
- Tests mit vitest für deterministische Hilfsfunktionen.
```

Copy

2 — `/speckit.specify`

> Briefing-Text:

```
Feature: askclaude (v1)

Als Nutzer will ich:
1. Auf der Startseite ein Textfeld sehen, in das ich eine Frage eingebe.
2. Nach Klick auf "Ask" soll Claudes Antwort TOKEN FÜR TOKEN erscheinen
   – nicht erst nach Ende der Generierung.
3. Während der Generierung ist der "Ask"-Button deaktiviert und zeigt
   "Generating…". Ein zweiter Button "Stop" bricht den Stream sofort ab.
4. Nach Abschluss steht die fertige Antwort vollständig im UI, der
   "Ask"-Button ist wieder aktiv.

Out of Scope für v1:
- Multi-Turn-Chat-Verlauf, Sessions, Auth, Persistenz.
- Tool-Use, Vision, Markdown-Rendering, Syntax-Highlighting.
- Rate-Limit-Handling jenseits von "Fehler im UI anzeigen".
```

Copy

3 — `/speckit.plan`

> ## Briefing-Text:

```
Plan für 001-askclaude.

Tech-Stack:
- Next.js 15 (App Router), React 19, TypeScript 5
- @anthropic-ai/sdk neuste stabile Version
- Runtime: nodejs (NICHT edge)
- Tailwind v4 (per default in create-next-app)
- vitest für Tests

Datei-Struktur:
- app/page.tsx                Server-Komponente, rendert <Chat />
- app/components/chat.tsx     Client-Komponente, Form + State + Reader
- app/api/chat/route.ts       POST-Handler, SDK-Stream → ReadableStream
- lib/anthropic.ts            getClient()-Singleton
- lib/types.ts                ChatRequest

Streaming-Flow:
- Server: client.messages.stream({...}) → for-await über die Events,
  nur content_block_delta (text_delta) als plain UTF-8 Bytes in einen
  ReadableStream<Uint8Array> schreiben. Kein SSE-Wrapping nötig — wir
  unwrappen serverseitig schon zu reinem Text.
- Client: fetch('/api/chat', {signal}).then(r => r.body.getReader()),
  in einer while-Schleife dekodieren und an setAnswer((a)=>a+chunk) hängen.
- Abort: AbortController auf dem Client; im Route-Handler den
  ReadableStream.cancel()-Hook nutzen, um stream.controller.abort()
  aufzurufen.

Modell-Wahl: claude-haiku-4-5 (TTFT-optimiert für sichtbares Streaming).

Akzeptanzkriterien:
- npm run dev startet, http://localhost:3000 zeigt das Eingabefeld.
- "Erkläre TypeScript Generics in 2 Sätzen" → erstes Token < 2s sichtbar.
- "Stop" während des Streams beendet ihn ohne Fehler im Server-Log.
```

4 — `/speckit.tasks`

Erwartete Aufgabenliste (zum Live-Validieren):

```
T-01  npx create-next-app@latest . --ts --app --tailwind --no-src-dir
T-02  npm i @anthropic-ai/sdk; .env.local mit ANTHROPIC_API_KEY anlegen
T-03  lib/anthropic.ts: getClient()-Singleton + lib/types.ts
T-04  app/api/chat/route.ts: POST-Handler, SDK-Stream → ReadableStream
T-05  app/components/chat.tsx: Form + Reader-Loop + AbortController
T-06  app/page.tsx: Server-Komponente, rendert <Chat />
T-07  vitest-Test für Delta-Extraktor (Pure Function)
T-08  README mit Setup-Schritten + .env.local.example
```

Copy

Jeder Task hat ein Akzeptanzkriterium — das ist der Unterschied zu freiem Vibe-Coding.

5 — `/speckit.implement` (live nur ausgewählte Tasks)

Im Vortrag nur **T-04** und **T-05** live ausführen — das sind die beiden mit dem visuellen Aha:

- T-04: Stream-Endpoint entsteht. Kurz `curl -N -X POST localhost:3000/api/chat -d '{"prompt":"Hi"}'` zeigen → Tokens in der Shell.
- T-05: Client-Komponente entsteht. Browser refreshen → Tokens im UI.

Bewusst stoppen vor T-07/T-08 und betonen: Spec-Kit zwingt nicht zur Vollautomation. Pausieren, prüfen, weiter.

6 — Wenn live etwas schiefgeht (Plan B)

- **Internet weg / API 401:** Pre-baked Repo bereithalten, das nur noch `npm run dev` braucht. Vorab einmal trocken durchgespielt.
- **API rate-limited:** Modell auf `claude-haiku-4-5` festnageln — das ist im Plan oben schon so. Sonnet nur falls Haiku ausfällt.
- `messages.stream` **API hat sich geändert:** Fallback-Snippet aus Abschnitt 7 zeigen — funktionierender Code statt Live-Generierung.
- **Demo-Zeit schmilzt:** Schritte 4 + 5 weglassen. 1 + 2 + 3 reichen, um den Spec-Kit-Mehrwert sichtbar zu machen.

7 — Code-Skelett als Fallback

Falls Live-`/speckit.implement` ausfällt, diese drei Dateien direkt zeigen / committen:

`app/api/chat/route.ts`

```
import Anthropic from "@anthropic-ai/sdk";

export const runtime = "nodejs";

export async function POST(req: Request) {
  const { prompt } = (await req.json()) as { prompt: string };
  const client = new Anthropic();

  const stream = client.messages.stream({
    model: "claude-haiku-4-5",
    max_tokens: 1024,
    messages: [{ role: "user", content: prompt }],
  });

  const encoder = new TextEncoder();
  const body = new ReadableStream<Uint8Array>({
    async start(controller) {
      try {
        for await (const event of stream) {
          if (
            event.type === "content_block_delta" &&
            event.delta.type === "text_delta"
          ) {
            controller.enqueue(encoder.encode(event.delta.text));
          }
        }
        controller.close();
      } catch (err) {
        controller.error(err);
      }
```

Take-aways für den Vortrag

- **Constitution-Disziplin gegen Scope-Creep:** Wenn jemand mid-demo ruft *„Brauchen wir nicht auch LangChain?"* — Constitution sagt nein.
- **Plan vor Code** macht den Streaming-Flow explizit, bevor die erste Zeile `route.ts` getippt wird. Niemand muss später raten.
- **Partielle Implementation ist okay:** wir haben v1 scharf spezifiziert, Multi-Turn und Markdown bewusst weggelassen.
- **Meta-Pointe** für die Bühne: *„Wir nutzen Spec-Kit, um eine App zu bauen, die uns das SDK aus Phase 2 zeigt — Werkzeug-Stack im Werkzeug-Stack."*