Vercel AI SDK

Version ai 7.0 · @ai-sdk/react 4.0 · verified 2026-11-02

AI SDK 7 in Next.js — providers, streaming text, Zod structured output, tool loops, useChat message parts, chat persistence, errors and cost.

Contents

Mental model 2

Providers and models 2

generateText and streamText 4

Structured output with Zod 5

Tools and the tool loop 7

Streaming to the UI 9

useChat and message parts 10

Persisting chats 12

Errors, retries and cost 14

Further reading 15

Mental model

The AI SDK is one provider-agnostic API for text, structured output and tool calls, plus React hooks that render a stream as it arrives. The model always runs on the server — a Route Handler calls `streamText` and turns the result into a UI message stream — and `useChat` on the client only sends messages and renders the parts that stream back. Swapping providers changes one `model` value, not your code.

Providers and models

Form	Example
Gateway string	`"anthropic/claude-sonnet-5.5"`
Provider package	`anthropic("claude-sonnet-5-5")`
Gateway key	`AI_GATEWAY_API_KEY`
Runtime	Node.js 22+, ESM only

A plain string resolves through the global provider, which defaults to the Vercel AI Gateway. Install a provider package such as `@ai-sdk/anthropic` to call a

vendor directly with its own key. AI SDK 7 dropped CommonJS and Node 18/20.

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

const { text, usage } = await generateText({
  model: "anthropic/claude-sonnet-5.5",
  instructions: "Answer in one sentence.",
  prompt: "What is backpressure?",
});
console.log(text, usage.totalTokens);
```

> **Gotcha:** AI SDK 7 renamed `system` to `instructions` and rejects `{ role: "system" }` inside `messages` by default. Old persisted chats that contain system messages fail until you migrate them.

generateText and streamText

Function	Returns	Use for
`generateText`	Promise of the full result	Jobs, scripts, server actions
`streamText`	Result with streams, immediately	Anything a user watches
`result.textStream`	Text chunks only	Plain text output
`result.stream`	Every part: text, tools, errors	UI streams

Both take the same options. `streamText` starts the request and returns at once; consume a stream or await a promise property to drive it. In v7, `result.fullStream` became `result.stream`, and `usage`, `toolCalls` and `content` now cover every step — read `finalStep` for the last step alone.

```ts
import { streamText } from "ai";

const result = streamText({
  model: "anthropic/claude-sonnet-5.5",
  prompt: "Explain idempotency briefly.",
});
for await (const chunk of result.textStream) {
  process.stdout.write(chunk);
}
console.log(await result.usage); // all steps
```

Structured output with Zod

Output	Produces
`Output.object({ schema })`	One validated object
`Output.array({ element })`	A typed list
`Output.choice({ options })`	One of fixed strings

Pass `output` to `generateText` or `streamText`; the schema is sent to the model and also validates the reply, so `result.output` is typed. Add `.describe()` to fields —

descriptions reach the model. A reply that fails validation throws `AI_NoObjectGeneratedError`.

```
import { generateText, Output } from "ai";
import { z } from "zod";

const { output } = await generateText({
  model: "anthropic/claude-sonnet-5.5",
  output: Output.object({ schema: z.object({
    title: z.string(),
    tags: z.array(z.string()).describe("lower"),
  }) }),
  prompt: "Tag this post: Postgres row locks",
});
console.log(output.tags); // string[]
```

> **Note:** `generateObject` and `streamObject` are deprecated (since v6), and v7 removed `experimental_output`. Use the `output` option, which also combines with tools.

Tools and the tool loop

Setting	Default
`stopWhen` on `generateText` / `streamText`	`isStepCount(1)`
`stopWhen` on `ToolLoopAgent`	`isStepCount(20)`
`toolChoice`	`"auto"`
`toolApproval`	Not applicable (runs automatically)

A tool is a description, an `inputSchema` and an `execute` function. A step is one model call plus the tools it requested; with `stopWhen` the SDK feeds results back and calls the model again until it answers in text or the condition is met. `ToolLoopAgent` packages model, instructions and tools for reuse.

```
import { generateText, isStepCount, tool }
  from "ai";
```

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

const weather = tool({
  description: "Current temperature for a city",
  inputSchema: z.object({ city: z.string() }),
  execute: async ({ city }) => `${city}: 21C`,
});
const { text } = await generateText({
  model: "anthropic/claude-sonnet-5.5",
  tools: { weather },
  stopWhen: isStepCount(5),
  prompt: "Is it T-shirt weather in Lisbon?",
});
```

> **Gotcha:** Without `stopWhen`, `generateText` stops after the step that called the tool, so `text` is empty. Structured output also costs a step — budget for tools plus one.

Streaming to the UI

Helper	Job
`convertToModelMessages`	`UIMessage[]` to model format
`toUIMessageStream`	Model stream to UI message stream
`createUIMessageStreamResponse`	Stream to a `Response`
`maxDuration`	Next.js limit for the route

`useChat` posts `UIMessage`s — display format with IDs and parts. The route converts them for the model, then converts the model stream back into UI message chunks. In v7 these are standalone helpers; the old `result.toUIMessageStreamResponse()` still works but is deprecated.

```
import { convertToModelMessages as toModel,
  createUIMessageStreamResponse, streamText,
  toUIMessageStream, type UIMessage } from "ai";
export const maxDuration = 30;
```

```ts
export async function POST(req: Request) {
  const body: { messages: UIMessage[] } =
    await req.json();
  const result = streamText({
    model: "anthropic/claude-sonnet-5.5",
    messages: await toModel(body.messages),
  });
  return createUIMessageStreamResponse({
    stream: toUIMessageStream({
      stream: result.stream }) });
}
```

useChat and message parts

Returned	Meaning
`messages`	`UIMessage[]`, each with `parts`
`sendMessage({ text })`	Post a user message
`status`	`submitted`, `streaming`, `ready`, `error`
`stop()` / `regenerate()`	Abort / retry last answer

`useChat` posts to `/api/chat` by default and manages no input state — keep the text box in your own `useState`. Render `parts`, not a single content string: a message mixes `text`, `reasoning` and typed tool parts named `tool-<name>`, each with a state such as `input-streaming`, `output-available` or `output-error`.

```
"use client";
import { useChat } from "@ai-sdk/react";
export function Chat() {
  const { messages, sendMessage, status } =
    useChat();
  const go = () => sendMessage({ text: "Hi" });
  return (<div>
    {messages.map((m) => <p key={m.id}>
      {m.parts.map((p, i) => p.type === "text"
        ? <span key={i}>{p.text}</span> : null)}
    </p>)}
    <button disabled={status !== "ready"}
      onClick={go}>Send</button>
```

```
  </div>);
}
```

Persisting chats

Need	Use
Save after each answer	`onEnd` in `toUIMessageStream`
Keep IDs stable	`originalMessages`, `generateMessageId`
Survive tab close	`result.consumeStream()`
Load old data safely	`validateUIMessages`

Store `UIMessage[]`, not model messages: they keep IDs, parts and metadata the UI needs to re-render. Pass the incoming history as `originalMessages` so `onEnd` receives the full conversation including the new reply.

```
import { convertToModelMessages as toModel,
  streamText, toUIMessageStream,
  type UIMessage } from "ai";
declare function save(m: UIMessage[]): void;
```

```
async function reply(messages: UIMessage[]) {
  const result = streamText({
    model: "anthropic/claude-sonnet-5.5",
    messages: await toModel(messages),
  });
  result.consumeStream(); // no await
  return toUIMessageStream({
    stream: result.stream,
    originalMessages: messages,
    onEnd: ({ messages: all }) => save(all) });
}
```

> **Gotcha:** `streamText` applies backpressure, so when the browser disconnects the model stream is aborted and `onEnd` never saves. Call `consumeStream()` to run it to completion server-side.

Errors, retries and cost

Setting	Default or effect
`maxRetries`	2; `0` disables
`timeout` / `abortSignal`	Abort a slow call
`maxOutputTokens`	Caps reply length
`onError` (UI stream)	Unmask error text for the client
`result.usage`	Tokens across all steps

Errors in a UI stream reach the client masked as “An error occurred.” so provider details do not leak; pass `onError` to `toUIMessageStream` to choose what the user sees. On the client, `error` plus `regenerate()` gives a retry button. Bound cost with `maxOutputTokens`, a `stopWhen` step cap, and a timeout.

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


const r = await generateText({
  model: "anthropic/claude-sonnet-5.5",
```

```
  prompt: "Summarise this changelog.",
  maxOutputTokens: 300,
  maxRetries: 1,
  timeout: 10_000,
});
const { inputTokens, outputTokens } = r.usage;
```

Further reading

- AI SDK documentation
- Migrate AI SDK 6.x to 7.0
- Tools and tool calling
- Building agents
- Chatbot message persistence