AI SDK: A Comprehensive Guide to Building AI Applications

This book provides a complete guide to the AI SDK by Vercel, offering insights into building AI-powered applications with frameworks like React, Next.js, Vue, and Node.js. It covers fundamental concepts, getting started tips, core SDK functionalities, UI components, advanced features, and reference materials, making it an essential resource for developers looking to harness the power of AI.

Table of Contents

- Streaming Custom Data
- Error Handling
- Transport
- Reading UIMessage Streams
- Message Metadata
- Stream Protocols

AI SDK RSC

- AI SDK RSC

Advanced Topics

- Advanced

References

- Reference
- AI SDK Core
- AI SDK UI
- AI SDK RSC
- Stream Helpers
- AI SDK Errors

Migration Guides

- Migration Guides

Troubleshooting

- Troubleshooting

AI SDK

The AI SDK is the TypeScript toolkit designed to help developers build AI-powered applications and agents with React, Next.js, Vue, Svelte, Node.js, and more.

Why use the AI SDK?

Integrating large language models (LLMs) into applications is complicated and heavily dependent on the specific model provider you use.

The AI SDK standardizes integrating artificial intelligence (AI) models across supported providers. This enables developers to focus on building great AI applications, not waste time on technical details.

For example, here's how you can generate text with various models using the AI SDK:

```
import { generateText } from "ai"
import { xai } from "@ai-sdk/xai"

const { text } = await generateText({
  model: xai("grok-3-beta"),
  prompt: "What is love?"
})
```

Love is a universal emotion that is characterized by feelings of affection, attachment, and warmth towards someone or something. It is a complex and multifaceted experience that can take many different forms, including romantic love, familial love, platonic love, and self-love.

The AI SDK has two main libraries:

- **AI SDK Core:** A unified API for generating text, structured objects, tool calls, and building agents with LLMs.
- **AI SDK UI:** A set of framework-agnostic hooks for quickly building chat and generative user interface.

Model Providers

The AI SDK supports multiple model providers.

Examples of Providers

- xAI Grok — Supports Image Input, Image Generation, Object Generation, Tool Usage, Tool Streaming
- OpenAI — Supports Image Input, Image Generation, Object Generation, Tool Usage, Tool Streaming
- Azure — Supports Image Input, Object Generation, Tool Usage, Tool Streaming
- Anthropic — Supports Image Input, Object Generation, Tool Usage, Tool Streaming
- Amazon Bedrock — Supports Image Input, Image Generation, Object Generation, Tool Usage, Tool Streaming
- Groq — Supports Object Generation, Tool Usage, Tool Streaming
- Fal AI — Supports Image Generation
- DeepInfra — Supports Image Input, Object Generation, Tool Usage, Tool Streaming
- Google Generative AI — Supports Image Input, Object Generation, Tool Usage, Tool Streaming

- Google Vertex AI — Supports Image Input, Image Generation, Object Generation, Tool Usage, Tool Streaming
- Mistral — Supports Image Input, Object Generation, Tool Usage, Tool Streaming
- Together.ai — Supports Object Generation, Tool Usage, Tool Streaming
- Cohere — Supports Tool Usage, Tool Streaming
- Fireworks — Supports Image Generation, Object Generation, Tool Usage, Tool Streaming
- DeepSeek — Supports Object Generation, Tool Usage, Tool Streaming
- Cerebras — Supports Object Generation, Tool Usage, Tool Streaming
- Perplexity
- Luma AI — Supports Image Generation

Templates

We've built some templates that include AI SDK integrations for different use cases, providers, and frameworks. You can use these templates to get started with your AI-powered application.

Starter Kits

- Chatbot Starter Template
  Uses the AI SDK and Next.js. Features persistence, multi-modal chat, and more.

- Internal Knowledge Base (RAG)
  Uses AI SDK Language Model Middleware for RAG and enforcing guardrails.

- Multi-Modal Chat
  Uses Next.js and AI SDK useChat hook for multi-modal message chat interface.

- Semantic Image Search
  An AI semantic image search app template built with Next.js, AI SDK, and Postgres.

- Natural Language PostgreSQL
  Query PostgreSQL using natural language with AI SDK and GPT-4o.

Feature Exploration

- Feature Flags Example
  AI SDK with Next.js, Feature Flags, and Edge Config for dynamic model switching.

- Chatbot with Telemetry
  AI SDK chatbot with OpenTelemetry support.

- Structured Object Streaming
  Uses AI SDK useObject hook to stream structured object generation.

- Multi-Step Tools
  Uses AI SDK streamText function to handle multiple tool steps automatically.

Frameworks

- Next.js OpenAI Starter
  Uses OpenAI GPT-4, AI SDK, and Next.js.

- Nuxt OpenAI Starter
  Uses OpenAI GPT-4, AI SDK, and Nuxt.js.

- SvelteKit OpenAI Starter
  Uses OpenAI GPT-4, AI SDK, and SvelteKit.

- Solid OpenAI Starter
  Uses OpenAI GPT-4, AI SDK, and Solid.

Generative UI

- Gemini Chatbot
  Uses Google Gemini, AI SDK, and Next.js.

- Generative UI with RSC (experimental)
  Uses Next.js, AI SDK, and streamUI to create generative UIs with React Server Components.

Security

- Bot Protection
  Uses Kasada, OpenAI GPT-4, AI SDK, and Next.js.

- Rate Limiting
  Uses Vercel KV, OpenAI GPT-4, AI SDK, and Next.js.

Join our Community

If you have questions about anything related to the AI SDK, you're always welcome to ask our community on GitHub Discussions.

`llms.txt` (for Cursor, Windsurf, Copilot, Claude etc.)

You can access the entire AI SDK documentation in Markdown format at ai-sdk.dev/llms.txt. This can be used to ask any LLM (assuming it has a big enough context window) questions about the AI SDK based on the most up-to-date documentation.

Example Usage

For instance, to prompt an LLM with questions about the AI SDK:

1. Copy the documentation contents from ai-sdk.dev/llms.txt
2. Use the following prompt format:

```
Documentation:
{paste documentation here}
---
Based on the above documentation, answer the following:
{your question}
```

Resources

- Docs
- Cookbook
- Providers
- Showcase
- GitHub
- Discussions

More

- Playground
- Contact Sales

About Vercel

- Next.js + Vercel
- Open Source Software
- GitHub
- X

Legal

- Privacy Policy

Overview

This page is a beginner-friendly introduction to high-level artificial intelligence (AI) concepts. To dive right into implementing the AI SDK, feel free to skip ahead to our quickstarts or learn about our supported models and providers.

The AI SDK standardizes integrating artificial intelligence (AI) models across supported providers. This enables developers to focus on building great AI applications, not waste time on technical details.

For example, here's how you can generate text with various models using the AI SDK:

```
import { generateText } from "ai"
import { xai } from "@ai-sdk/xai"

const { text } = await generateText({
  model: xai("grok-3-beta"),
  prompt: "What is love?"
})
```

Love is a universal emotion that is characterized by feelings of affection, attachment, and warmth towards someone or something. It is a complex and multifaceted experience that can take many different forms, including romantic love, familial love, platonic love, and self-love.

To effectively leverage the AI SDK, it helps to familiarize yourself with the following concepts:

Generative Artificial Intelligence

Generative artificial intelligence refers to models that predict and generate various types of outputs (such as text, images, or audio) based on what's statistically likely, pulling from patterns they've learned from their training data. For example:

- Given a photo, a generative model can generate a caption.
- Given an audio file, a generative model can generate a transcription.
- Given a text description, a generative model can generate an image.

Large Language Models

A **large language model (LLM)** is a subset of generative models focused primarily on **text**. An LLM takes a sequence of words as input and aims to predict the most likely sequence to follow. It assigns probabilities to potential next sequences and then selects one. The model continues to generate sequences until it meets a specified stopping criterion.

LLMs learn by training on massive collections of written text, which means they will be better suited to some use cases than others. For example, a model trained on GitHub data would understand the probabilities of sequences in source code particularly well.

However, it's crucial to understand LLMs' limitations. When asked about less known or absent information, like the birthday of a personal relative, LLMs might "hallucinate" or make up information. It's essential to consider how well-represented the information you need is in the model.

Embedding Models

An **embedding model** is used to convert complex data (like words or images) into a dense vector (a list of numbers) representation, known as an embedding. Unlike generative models, embedding models do not generate new text or data. Instead, they provide representations of semantic and syntactic relationships between entities that can be used as input for other models or other natural language processing tasks.

In the next section, you will learn about the difference between models providers and models, and which ones are available in the AI SDK.

Providers and Models

Companies such as OpenAI and Anthropic (providers) offer access to a range of large language models (LLMs) with differing strengths and capabilities through their own APIs.

Each provider typically has its own unique method for interfacing with their models, complicating the process of switching providers and increasing the risk of vendor lock-in.

To solve these challenges, AI SDK Core offers a standardized approach to interacting with LLMs through a language model specification that abstracts differences between providers. This unified interface allows you to switch between providers with ease while using the same API for all providers.

Here is an overview of the AI SDK Provider Architecture:

AI SDK Providers

The AI SDK comes with a wide range of providers that you can use to interact with different language models:

- xAI Grok Provider (`@ai-sdk/xai`)
- OpenAI Provider (`@ai-sdk/openai`)
- Azure OpenAI Provider (`@ai-sdk/azure`)
- Anthropic Provider (`@ai-sdk/anthropic`)
- Amazon Bedrock Provider (`@ai-sdk/amazon-bedrock`)
- Google Generative AI Provider (`@ai-sdk/google`)
- Google Vertex Provider (`@ai-sdk/google-vertex`)
- Mistral Provider (`@ai-sdk/mistral`)
- Together.ai Provider (`@ai-sdk/togetherai`)
- Cohere Provider (`@ai-sdk/cohere`)
- Fireworks Provider (`@ai-sdk/fireworks`)
- DeepInfra Provider (`@ai-sdk/deepinfra`)
- DeepSeek Provider (`@ai-sdk/deepseek`)
- Cerebras Provider (`@ai-sdk/cerebras`)
- Groq Provider (`@ai-sdk/groq`)
- Perplexity Provider (`@ai-sdk/perplexity`)
- ElevenLabs Provider (`@ai-sdk/elevenlabs`)
- LMNT Provider (`@ai-sdk/lmnt`)
- Hume Provider (`@ai-sdk/hume`)
- Rev.ai Provider (`@ai-sdk/revai`)
- Deepgram Provider (`@ai-sdk/deepgram`)
- Gladia Provider (`@ai-sdk/gladia`)
- LMNT Provider (`@ai-sdk/lmnt`)
- AssemblyAI Provider (`@ai-sdk/assemblyai`)

You can also use the OpenAI Compatible provider with OpenAI-compatible APIs:

- LM Studio
- Baseten
- Heroku

Our language model specification is published as an open-source package, which you can use to create custom providers.

The open-source community has created the following providers:

- Ollama Provider (`ollama-ai-provider`)
- FriendliAI Provider (`@friendliai/ai-provider`)
- Portkey Provider (`@portkey-ai/vercel-provider`)
- Cloudflare Workers AI Provider (`workers-ai-provider`)
- OpenRouter Provider (`@openrouter/ai-sdk-provider`)
- Requesty Provider (`@requesty/ai-sdk`)
- Crosshatch Provider (`@crosshatch/ai-provider`)
- Mixedbread Provider (`mixedbread-ai-provider`)
- Voyage AI Provider (`voyage-ai-provider`)
- Mem0 Provider (`@mem0/vercel-ai-provider`)
- Letta Provider (`@letta-ai/vercel-ai-sdk-provider`)
- Spark Provider (`spark-ai-provider`)
- AnthropicVertex Provider (`anthropic-vertex-ai`)
- LangDB Provider (`@langdb/vercel-provider`)
- Dify Provider (`dify-ai-provider`)
- Sarvam Provider (`sarvam-ai-provider`)
- Claude Code Provider (`ai-sdk-provider-claude-code`)
- Built-in AI Provider (`built-in-ai`)
- Gemini CLI Provider (`ai-sdk-provider-gemini-cli`)
- A2A Provider (`a2a-ai-provider`)
- SAP-AI Provider (`@mymediset/sap-ai-provider`)

Self-Hosted Models

You can access self-hosted models with the following providers:

- Ollama Provider
- LM Studio
- Baseten

Additionally, any self-hosted provider that supports the OpenAI specification can be used with the OpenAI Compatible Provider.

Model Capabilities

The AI providers support different language models with various capabilities.
Here are the capabilities of popular models:

Provider	Model	Image Input	Object Generation	Tool Usage	Tool Streaming
xAI Grok	`grok-4`				
xAI Grok	`grok-3`				
xAI Grok	`grok-3-fast`				
xAI Grok	`grok-3-mini`				
xAI Grok	`grok-3-mini-fast`				
xAI Grok	`grok-2-1212`				
xAI Grok	`grok-2-vision-1212`				
xAI Grok	`grok-beta`				
xAI Grok	`grok-vision-beta`				

Provider	Model	Image Input	Object Generation	Tool Usage	Tool Streaming
Vercel	`v0-1.0-md`				
OpenAI	`gpt-4.1`				
OpenAI	`gpt-4.1-mini`				
OpenAI	`gpt-4.1-nano`				
OpenAI	`gpt-4o`				
OpenAI	`gpt-4o-mini`				
OpenAI	`gpt-4.1`				
OpenAI	`gpt-4`				
OpenAI	`o3-mini`				
OpenAI	`o3`				
OpenAI	`o4-mini`				
OpenAI	`o1`				
Anthropic	`claude-opus-4-latest`				
Anthropic	`claude-sonnet-4-latest`				
Anthropic	`claude-3-7-sonnet-latest`				
Anthropic	`claude-3-5-sonnet-latest`				
Anthropic	`claude-3-5-sonnet-latest`				
Anthropic	`claude-3-5-haiku-latest`				
Mistral	`pixtral-large-latest`				
Mistral	`mistral-large-latest`				
Mistral	`mistral-medium-latest`				
Mistral	`mistral-medium-2505`				
Mistral	`mistral-small-latest`				
Mistral	`pixtral-12b-2409`				
Google Generative AI	`gemini-2.0-flash-exp`				
Google Generative AI	`gemini-1.5-flash`				
Google Generative AI	`gemini-1.5-pro`				
Google Vertex	`gemini-2.0-flash-exp`				
Google Vertex	`gemini-1.5-flash`				
Google Vertex	`gemini-1.5-pro`				
DeepSeek	`deepseek-chat`				
DeepSeek	`deepseek-reasoner`				
Cerebras	`llama3.1-8b`				
Cerebras	`llama3.1-70b`				
Cerebras	`llama3.3-70b`				
Groq	`meta-llama/llama-4-scout-17b-16e-instruct`				
Groq	`llama-3.3-70b-versatile`				
Groq	`llama-3.1-8b-instant`				
Groq	`mixtral-8x7b-32768`				
Groq	`gemma2-9b-it`				

This table is not exhaustive. Additional models can be found in the provider documentation pages and on the provider websites.

Prompts

Prompts are instructions that you give a large language model (LLM) to tell it what to do.
It's like when you ask someone for directions; the clearer your question, the better the directions you'll get.

Many LLM providers offer complex interfaces for specifying prompts. They involve different roles and message types.
While these interfaces are powerful, they can be hard to use and understand.

In order to simplify prompting, the AI SDK supports text, message, and system prompts.

Text Prompts

Text prompts are strings.
They are ideal for simple generation use cases,
e.g. repeatedly generating content for variants of the same prompt text.

You can set text prompts using the `prompt` property made available by AI SDK functions like `streamText` or `generateObject`.
You can structure the text in any way and inject variables, e.g. using a template literal.

```
const result = await generateText({
  model: 'openai/gpt-4.1',
  prompt: 'Invent a new holiday and describe its traditions.',
});
```

You can also use template literals to provide dynamic data to your prompt.

```
const result = await generateText({
  model: 'openai/gpt-4.1',
  prompt:
    `I am planning a trip to ${destination} for ${lengthOfStay} days. ` +
    `Please suggest the best tourist activities for me to do.`,
});
```

System Prompts

System prompts are the initial set of instructions given to models that help guide and constrain the models' behaviors and responses.
You can set system prompts using the `system` property.
System prompts work with both the `prompt` and the `messages` properties.

```
const result = await generateText({
  model: 'openai/gpt-4.1',
  system:
    `You help planning travel itineraries. ` +
    `Respond to the users' request with a list ` +
    `of the best stops to make in their destination.`,
  prompt:
    `I am planning a trip to ${destination} for ${lengthOfStay} days. ` +
    `Please suggest the best tourist activities for me to do.`,
});
```

When you use a message prompt, you can also use system messages instead of a system prompt.

Message Prompts

A message prompt is an array of user, assistant, and tool messages.
They are great for chat interfaces and more complex, multi-modal prompts.
You can use the `messages` property to set message prompts.

Each message has a `role` and a `content` property. The content can either be text (for user and assistant messages), or an array of relevant parts (data) for that message type.

```
const result = await generateText({
  model: 'openai/gpt-4.1',
  messages: [
    { role: 'user', content: 'Hi!' },
    { role: 'assistant', content: 'Hello, how can I help?' },
    { role: 'user', content: 'Where can I buy the best Currywurst in Berlin?' },
  ],
});
```

Instead of sending a text in the `content` property, you can send an array of parts that includes a mix of text and other content parts.

> Not all language models support all message and content types. For example, some models might not be capable of handling multi-modal inputs or tool messages. Learn more about the capabilities of select models.

Provider Options

You can pass through additional provider-specific metadata to enable provider-specific functionality at 3 levels.

Function Call Level

Functions like `streamText` or `generateText` accept a `providerOptions` property.

Adding provider options at the function call level should be used when you do not need granular control over where the provider options are applied.

```
const { text } = await generateText({
  model: azure('your-deployment-name'),
  providerOptions: {
    openai: {
      reasoningEffort: 'low',
    },
  },
});
```

Message Level

For granular control over applying provider options at the message level, you can pass `providerOptions` to the message object:

```
import { ModelMessage } from 'ai';
```

```
const messages: ModelMessage[] = [
  {
    role: 'system',
    content: 'Cached system message',
    providerOptions: {
      // Sets a cache control breakpoint on the system message
      anthropic: { cacheControl: { type: 'ephemeral' } },
    },
  },
];
```

Message Part Level

Certain provider-specific options require configuration at the message part level:

```
import { ModelMessage } from 'ai';

const messages: ModelMessage[] = [
  {
    role: 'user',
    content: [
      {
        type: 'text',
        text: 'Describe the image in detail.',
        providerOptions: {
          openai: { imageDetail: 'low' },
        },
      },
      {
        type: 'image',
        image: 'https://github.com/vercel/ai/blob/main/examples/ai-core/data/comi
        // Sets image detail configuration for image part:
        providerOptions: {
          openai: { imageDetail: 'low' },
        },
      },
    ],
  },
];
```

> AI SDK UI hooks like `useChat` return arrays of `UIMessage` objects, which do not support provider options. We recommend using the `convertToModelMessages` function to convert `UIMessage` objects to `ModelMessage` objects before applying or appending message(s) or message parts with `providerOptions`.

User Messages

Text Parts

Text content is the most common type of content. It is a string that is passed to the model.

If you only need to send text content in a message, the `content` property can be a string, but you can also use it to send multiple content parts.

```
const result = await generateText({
  model: 'openai/gpt-4.1',
  messages: [
    {
      role: 'user',
      content: [
        {
          type: 'text',
          text: 'Where can I buy the best Currywurst in Berlin?',
        },
      ],
    },
  ],
});
```

Image Parts

User messages can include image parts. An image can be one of the following:

- base64-encoded image:
  - `string` with base-64 encoded content
  - data URL `string`, e.g. `...`
- binary image:
  - `ArrayBuffer`
  - `Uint8Array`
  - `Buffer`
- URL:
  - http(s) URL `string`, e.g. `https://example.com/image.png`
  - `URL` object, e.g. `new URL('https://example.com/image.png')`

Example: Binary image (Buffer)

```
const result = await generateText({
  model,
  messages: [
    {
      role: 'user',
      content: [
        { type: 'text', text: 'Describe the image in detail.' },
        {
          type: 'image',
          image: fs.readFileSync('./data/comic-cat.png'),
        },
      ],
    },
  ],
});
```

Example: Base-64 encoded image (string)

```
const result = await generateText({
  model: 'openai/gpt-4.1',
```

```
  messages: [
    {
      role: 'user',
      content: [
        { type: 'text', text: 'Describe the image in detail.' },
        {
          type: 'image',
          image: fs.readFileSync('./data/comic-cat.png').toString('base64'),
        },
      ],
    },
  ],
});
```

Example: Image URL (string)

```
const result = await generateText({
  model: 'openai/gpt-4.1',
  messages: [
    {
      role: 'user',
      content: [
        { type: 'text', text: 'Describe the image in detail.' },
        {
          type: 'image',
          image: 'https://github.com/vercel/ai/blob/main/examples/ai-core/data/co
        },
      ],
    },
  ],
});
```

File Parts

Only a few providers and models currently support file parts: Google Generative AI, Google Vertex AI, OpenAI (for `wav` and `mp3` audio with `gpt-4o-audio-preview`), Anthropic, OpenAI (for `pdf`).

User messages can include file parts. A file can be one of the following:

- base64-encoded file:
  - `string` with base-64 encoded content
  - data URL `string`, e.g. `...`
- binary data:
  - `ArrayBuffer`
  - `Uint8Array`
  - `Buffer`
- URL:
  - http(s) URL `string`, e.g. `https://example.com/some.pdf`
  - `URL` object, e.g. `new URL('https://example.com/some.pdf')`

You need to specify the MIME type of the file you are sending.

Example: PDF file from Buffer

```
import { google } from '@ai-sdk/google';
import { generateText } from 'ai';

const result = await generateText({
  model: google('gemini-1.5-flash'),
  messages: [
    {
      role: 'user',
      content: [
        { type: 'text', text: 'What is the file about?' },
        {
          type: 'file',
          mediaType: 'application/pdf',
          data: fs.readFileSync('./data/example.pdf'),
          filename: 'example.pdf', // optional, not used by all providers
        },
      ],
    },
  ],
});
```

Example: mp3 audio file from Buffer

```
import { openai } from '@ai-sdk/openai';
import { generateText } from 'ai';

const result = await generateText({
  model: openai('gpt-4o-audio-preview'),
  messages: [
    {
      role: 'user',
      content: [
        { type: 'text', text: 'What is the audio saying?' },
        {
          type: 'file',
          mediaType: 'audio/mpeg',
          data: fs.readFileSync('./data/galileo.mp3'),
        },
      ],
    },
  ],
});
```

Assistant Messages

Assistant messages are messages that have a role of `assistant`.
They are typically previous responses from the assistant
and can contain text, reasoning, and tool call parts.

Example: Assistant message with text content

```
const result = await generateText({
  model: 'openai/gpt-4.1',
  messages: [
```

```
    { role: 'user', content: 'Hi!' },
    { role: 'assistant', content: 'Hello, how can I help?' },
  ],
});
```

Example: Assistant message with text content in array

```
const result = await generateText({
  model: 'openai/gpt-4.1',
  messages: [
    { role: 'user', content: 'Hi!' },
    { role: 'assistant', content: [{ type: 'text', text: 'Hello, how can I help?'
  ],
});
```

Example: Assistant message with tool call content

```
const result = await generateText({
  model: 'openai/gpt-4.1',
  messages: [
    { role: 'user', content: 'How many calories are in this block of cheese?' },
    {
      role: 'assistant',
      content: [
        {
          type: 'tool-call',
          toolCallId: '12345',
          toolName: 'get-nutrition-data',
          input: { cheese: 'Roquefort' },
        },
      ],
    },
  ],
});
```

Example: Assistant message with file content

This content part is for model-generated files. Only a few models support this, and only for file types that they can generate.

```
const result = await generateText({
  model: 'openai/gpt-4.1',
  messages: [
    { role: 'user', content: 'Generate an image of a roquefort cheese!' },
    {
      role: 'assistant',
      content: [
        {
          type: 'file',
          mediaType: 'image/png',
          data: fs.readFileSync('./data/roquefort.jpg'),
        },
      ],
    },
```

```
  ],
});
```

Tool messages

Tools (also known as function calling) are programs
that you can provide an LLM to extend its built-in functionality. This can be
anything from calling an external API to calling functions within your UI.
Learn more about Tools in the next section.

For models that support tool calls, assistant messages can contain tool call parts, and tool messages can contain tool output parts.
A single assistant message can call multiple tools, and a single tool message can contain multiple tool results.

```
const result = await generateText({
  model: 'openai/gpt-4.1',
  messages: [
    {
      role: 'user',
      content: [
        { type: 'text', text: 'How many calories are in this block of cheese?' },
        { type: 'image', image: fs.readFileSync('./data/roquefort.jpg') },
      ],
    },
    {
      role: 'assistant',
      content: [
        {
          type: 'tool-call',
          toolCallId: '12345',
          toolName: 'get-nutrition-data',
          input: { cheese: 'Roquefort' },
        },
        // there could be more tool calls here (parallel calling)
      ],
    },
    {
      role: 'tool',
      content: [
        {
          type: 'tool-result',
          toolCallId: '12345', // needs to match the tool call id
          toolName: 'get-nutrition-data',
          output: {
            type: 'json',
            value: {
              name: 'Cheese, roquefort',
              calories: 369,
              fat: 31,
              protein: 22,
            },
          },
        },
        // there could be more tool results here (parallel calling)
```

```
      ],
    },
  ],
});
```

Multi-modal Tool Results

Multi-part tool results are experimental and only supported by Anthropic.

Tool results can be multi-part and multi-modal, e.g. a text and an image.
You can use the `experimental_content` property on tool parts to specify multi-part tool results.

```
const result = await generateText({
  model: 'openai/gpt-4.1',
  messages: [
    // ...
    {
      role: 'tool',
      content: [
        {
          type: 'tool-result',
          toolCallId: '12345', // needs to match the tool call id
          toolName: 'get-nutrition-data',
          // for models that do not support multi-part tool results,
          // you can include a regular output part:
          output: {
            type: 'json',
            value: {
              name: 'Cheese, roquefort',
              calories: 369,
              fat: 31,
              protein: 22,
            },
          },
        },
        {
          type: 'tool-result',
          toolCallId: '12345', // needs to match the tool call id
          toolName: 'get-nutrition-data',
          // for models that support multi-part tool results,
          // you can include a multi-part content part:
          output: {
            type: 'content',
            value: [
              { type: 'text', text: 'Here is an image of the nutrition data for t
              {
                type: 'media',
                data: fs
                  .readFileSync('./data/roquefort-nutrition-data.png')
                  .toString('base64'),
                mediaType: 'image/png',
              },
            ],
          },
        },
```

```
      ],
    },
  ],
});
```

System Messages

System messages are messages that are sent to the model before the user messages to guide the assistant's behavior.
You can alternatively use the `system` property.

```
const result = await generateText({
  model: 'openai/gpt-4.1',
  messages: [
    { role: 'system', content: 'You help planning travel itineraries.' },
    {
      role: 'user',
      content: 'I am planning a trip to Berlin for 3 days. Please suggest the bes
    },
  ],
});
```

Navigation

Previous: Providers and Models | Next: Tools

Tools

While large language models (LLMs) have incredible generation capabilities,
they struggle with discrete tasks (e.g. mathematics) and interacting with the outside world (e.g. getting the weather).

Tools are actions that an LLM can invoke.
The results of these actions can be reported back to the LLM to be considered in the next response.

For example, when you ask an LLM for the "weather in London", and there is a weather tool available, it could call a tool
with London as the argument. The tool would then fetch the weather data and return it to the LLM. The LLM can then use this
information in its response.

What is a tool?

A tool is an object that can be called by the model to perform a specific task.
You can use tools with `generateText`
and `streamText` by passing one or more tools to the `tools` parameter.

A tool consists of three properties:

- **`description`**: An optional description of the tool that can influence when the tool is picked.
- **`inputSchema`**: A Zod schema or a JSON schema that defines the input required for the tool to run. The schema is consumed by the LLM, and also used to validate the LLM tool calls.
- **`execute`**: An optional async function that is called with the arguments from the tool call.

`streamUI` uses UI generator tools with a `generate` function that can return
React components.

If the LLM decides to use a tool, it will generate a tool call.
Tools with an `execute` function are run automatically when these calls are generated.
The output of the tool calls are returned using tool result objects.

You can automatically pass tool results back to the LLM
using multi-step calls with `streamText` and `generateText`.

Schemas

Schemas are used to define the parameters for tools and to validate the tool calls.

The AI SDK supports both raw JSON schemas (using the `jsonSchema` function)
and Zod schemas (either directly or using the `zodSchema` function).

Zod is a popular TypeScript schema validation library.
You can install it with:

```
pnpm add zod
```

You can then specify a Zod schema, for example:

```
import z from 'zod';
```

```
const recipeSchema = z.object({
  recipe: z.object({
    name: z.string(),
    ingredients: z.array(
      z.object({
        name: z.string(),
        amount: z.string(),
      }),
    ),
    steps: z.array(z.string()),
  }),
});
```

You can also use schemas for structured output generation with `generateObject` and `streamObject`.

Toolkits

When you work with tools, you typically need a mix of application specific tools and general purpose tools.
There are several providers that offer pre-built tools as **toolkits** that you can use out of the box:

- **agentic** - A collection of 20+ tools. Most tools connect to access external APIs such as Exa or E2B.
- **browserbase** - Browser tool that runs a headless browser
- **browserless** - Browser automation service with AI integration - self hosted or cloud based
- **Smithery** - Smithery provides an open marketplace of 6K+ MCPs, including Browserbase and Exa.
- **Stripe agent tools** - Tools for interacting with Stripe.
- **StackOne ToolSet** - Agentic integrations for hundreds of enterprise SaaS
- **Toolhouse** - AI function-calling in 3 lines of code for over 25 different actions.
- **Agent Tools** - A collection of tools for agents.
- **AI Tool Maker** - A CLI utility to generate AI SDK tools from OpenAPI specs.
- **Composio** - Composio provides 250+ tools like GitHub, Gmail, Salesforce and more.
- **Interlify** - Convert APIs into tools so that AI can connect to your backend in minutes.
- **Freestyle** - Tool for your AI to execute JavaScript or TypeScript with arbitrary node modules.
- **JigsawStack** - JigsawStack provides over 30+ small custom fine tuned models available for specific uses.

Do you have open source tools or tool libraries that are compatible with the AI SDK? Please file a pull request to add them to this list.

Learn more

The AI SDK Core Tool Calling
and Agents documentation has more information about tools and tool calling.

Previous: Prompts | Next: Streaming

Streaming

Streaming conversational text UIs (like ChatGPT) have gained massive popularity over the past few months. This section explores the benefits and drawbacks of streaming and blocking interfaces.

Large language models (LLMs) are extremely powerful. However, when generating long outputs, they can be very slow compared to the latency you're likely used to. If you try to build a traditional blocking UI, your users might easily find themselves staring at loading spinners for 5, 10, even up to 40s waiting for the entire LLM response to be generated. This can lead to a poor user experience, especially in conversational applications like chatbots. Streaming UIs can help mitigate this issue by **displaying parts of the response as they become available**.

Blocking UI
Blocking responses wait until the full response is available before displaying it.

Streaming UI
Streaming responses can transmit parts of the response as they become available.

Real-world Examples

Here are 2 examples that illustrate how streaming UIs can improve user experiences in a real-world setting – the first uses a blocking UI, while the second uses a streaming UI.

Blocking UI

Generate
...

Streaming UI

Generate
...

As you can see, the streaming UI is able to start displaying the response much faster than the blocking UI. This is because the blocking UI has to wait for the entire response to be generated before it can display anything, while the streaming UI can display parts of the response as they become available.

While streaming interfaces can greatly enhance user experiences, especially with larger language models, they aren't always necessary or beneficial. If you can achieve your desired functionality using a smaller, faster model without resorting to streaming, this route can often lead to simpler and more manageable development processes.

However, regardless of the speed of your model, the AI SDK is designed to make implementing streaming UIs as simple as possible. In the example below, we stream text generation from OpenAI's `gpt-4.1` in under 10 lines of code using the SDK's `streamText` function:

```
import { openai } from '@ai-sdk/openai';
import { streamText } from 'ai';

const { textStream } = streamText({
  model: openai('gpt-4.1'),
  prompt: 'Write a poem about embedding models.',
});
```

```
for await (const textPart of textStream) {
  console.log(textPart);
}
```

For an introduction to streaming UIs and the AI SDK, check out our Getting Started guides.

On this page

Agents

When building AI applications, you often need **systems that can understand context and take meaningful actions**. When building these systems, the key consideration is finding the right balance between flexibility and control. Let's explore different approaches and patterns for building these systems, with a focus on helping you match capabilities to your needs.

Building Blocks

When building AI systems, you can combine these fundamental components:

Single-Step LLM Generation

The basic building block - one call to an LLM to get a response. Useful for straightforward tasks like classification or text generation.

Tool Usage

Enhanced capabilities through tools (like calculators, APIs, or databases) that the LLM can use to accomplish tasks. Tools provide a controlled way to extend what the LLM can do.

When solving complex problems, **an LLM can make multiple tool calls across multiple steps without you explicitly specifying the order** - for example, looking up information in a database, using that to make calculations, and then storing results. The AI SDK makes this multi-step tool usage straightforward through the `stopWhen` parameter.

Multi-Agent Systems

Multiple LLMs working together, each specialized for different aspects of a complex task. This enables sophisticated behaviors while keeping individual components focused.

Patterns

These building blocks can be combined with workflow patterns that help manage complexity:

- Sequential Processing - Steps executed in order
- Parallel Processing - Independent tasks run simultaneously
- Evaluation/Feedback Loops - Results checked and improved iteratively
- Orchestration - Coordinating multiple components
- Routing - Directing work based on context

Choosing Your Approach

The key factors to consider:

- **Flexibility vs Control** - How much freedom does the LLM need vs how tightly must you constrain its actions?
- **Error Tolerance** - What are the consequences of mistakes in your use case?
- **Cost Considerations** - More complex systems typically mean more LLM calls and higher costs
- **Maintenance** - Simpler architectures are easier to debug and modify

Start with the simplest approach that meets your needs. Add complexity only when required by:

1. Breaking down tasks into clear steps
2. Adding tools for specific capabilities
3. Implementing feedback loops for quality control
4. Introducing multiple agents for complex workflows

Let's look at examples of these patterns in action.

Patterns with Examples

The following patterns, adapted from Anthropic's guide on building effective agents, serve as building blocks that can be combined to create comprehensive workflows. Each pattern addresses specific aspects of task execution, and by combining them thoughtfully, you can build reliable solutions for complex problems.

Sequential Processing (Chains)

The simplest workflow pattern executes steps in a predefined order. Each step's output becomes input for the next step, creating a clear chain of operations. This pattern is ideal for tasks with well-defined sequences, like content generation pipelines or data transformation processes.

```
import { openai } from '@ai-sdk/openai';

import { generateText, generateObject } from 'ai';

import { z } from 'zod';

async function generateMarketingCopy(input: string) {
  const model = openai('gpt-4o');

  // First step: Generate marketing copy
  const { text: copy } = await generateText({
    model,
    prompt: `Write persuasive marketing copy for: ${input}. Focus on benefits and
  });

  // Perform quality check on copy
  const { object: qualityMetrics } = await generateObject({
    model,
    schema: z.object({
      hasCallToAction: z.boolean(),
      emotionalAppeal: z.number().min(1).max(10),
      clarity: z.number().min(1).max(10),
    }),
    prompt: `Evaluate this marketing copy for:

    1. Presence of call to action (true/false)
    2. Emotional appeal (1-10)
    3. Clarity (1-10)

    Copy to evaluate: ${copy}`,
  });

  // If quality check fails, regenerate with more specific instructions
  if (
```

```
    !qualityMetrics.hasCallToAction ||
    qualityMetrics.emotionalAppeal < 7 ||
    qualityMetrics.clarity < 7
  ) {
    const { text: improvedCopy } = await generateText({
      model,
      prompt: `Rewrite this marketing copy with:
      ${!qualityMetrics.hasCallToAction ? '- A clear call to action' : ''}
      ${qualityMetrics.emotionalAppeal < 7 ? '- Stronger emotional appeal' : ''}
      ${qualityMetrics.clarity < 7 ? '- Improved clarity and directness' : ''}

      Original copy: ${copy}`,
    });

    return { copy: improvedCopy, qualityMetrics };
  }

  return { copy, qualityMetrics };
}
```

Routing

This pattern allows the model to make decisions about which path to take through a workflow based on context and intermediate results. The model acts as an intelligent router, directing the flow of execution between different branches of your workflow. This is particularly useful when handling varied inputs that require different processing approaches. In the example below, the results of the first LLM call change the properties of the second LLM call like model size and system prompt.

```
import { openai } from '@ai-sdk/openai';

import { generateObject, generateText } from 'ai';

import { z } from 'zod';

async function handleCustomerQuery(query: string) {
  const model = openai('gpt-4o');

  // First step: Classify the query type
  const { object: classification } = await generateObject({
    model,
    schema: z.object({
      reasoning: z.string(),
      type: z.enum(['general', 'refund', 'technical']),
      complexity: z.enum(['simple', 'complex']),
    }),
    prompt: `Classify this customer query:
    ${query}

    Determine:
    1. Query type (general, refund, or technical)
    2. Complexity (simple or complex)
    3. Brief reasoning for classification`,
  });
```

```
  // Route based on classification
  // Set model and system prompt based on query type and complexity
  const { text: response } = await generateText({
    model:
      classification.complexity === 'simple'
        ? openai('gpt-4o-mini')
        : openai('o3-mini'),
    system: {
      general: 'You are an expert customer service agent handling general inquiri
      refund:
        'You are a customer service agent specializing in refund requests. Follow
      technical:
        'You are a technical support specialist with deep product knowledge. Focu
    }[classification.type],
    prompt: query,
  });

  return { response, classification };
}
```

Parallel Processing

Some tasks can be broken down into independent subtasks that can be executed simultaneously. This pattern takes advantage of parallel execution to improve efficiency while maintaining the benefits of structured workflows. For example, analyzing multiple documents or processing different aspects of a single input concurrently (like code review).

```
import { openai } from '@ai-sdk/openai';

import { generateText, generateObject } from 'ai';

import { z } from 'zod';

// Example: Parallel code review with multiple specialized reviewers
async function parallelCodeReview(code: string) {
  const model = openai('gpt-4o');

  // Run parallel reviews
  const [securityReview, performanceReview, maintainabilityReview] = await Promis
    generateObject({
      model,
      system:
        'You are an expert in code security. Focus on identifying security vulner
      schema: z.object({
        vulnerabilities: z.array(z.string()),
        riskLevel: z.enum(['low', 'medium', 'high']),
        suggestions: z.array(z.string()),
      }),
      prompt: `Review this code:
      ${code}`,
    }),
    generateObject({
      model,
      system:
```

```
        'You are an expert in code performance. Focus on identifying performance
      schema: z.object({
        issues: z.array(z.string()),
        impact: z.enum(['low', 'medium', 'high']),
        optimizations: z.array(z.string()),
      }),
      prompt: `Review this code:
      ${code}`,
    }),
    generateObject({
      model,
      system:
        'You are an expert in code quality. Focus on code structure, readability,
      schema: z.object({
        concerns: z.array(z.string()),
        qualityScore: z.number().min(1).max(10),
        recommendations: z.array(z.string()),
      }),
      prompt: `Review this code:
      ${code}`,
    }),
  ]);

  const reviews = [
    { ...securityReview.object, type: 'security' },
    { ...performanceReview.object, type: 'performance' },
    { ...maintainabilityReview.object, type: 'maintainability' },
  ];

  // Aggregate results using another model instance
  const { text: summary } = await generateText({
    model,
    system: 'You are a technical lead summarizing multiple code reviews.',
    prompt: `Synthesize these code review results into a concise summary with key
    ${JSON.stringify(reviews, null, 2)}`,
  });

  return { reviews, summary };
}
```

Orchestrator-Worker

In this pattern, a primary model (orchestrator) coordinates the execution of specialized workers. Each worker is optimized for a specific subtask, while the orchestrator maintains overall context and ensures coherent results. This pattern excels at complex tasks requiring different types of expertise or processing.

```
import { openai } from '@ai-sdk/openai';

import { generateObject } from 'ai';

import { z } from 'zod';

async function implementFeature(featureRequest: string) {
```

```
  // Orchestrator: Plan the implementation
  const { object: implementationPlan } = await generateObject({
    model: openai('o3-mini'),
    schema: z.object({
      files: z.array(
        z.object({
          purpose: z.string(),
          filePath: z.string(),
          changeType: z.enum(['create', 'modify', 'delete']),
        }),
      ),
      estimatedComplexity: z.enum(['low', 'medium', 'high']),
    }),
    system: 'You are a senior software architect planning feature implementations
    prompt: `Analyze this feature request and create an implementation plan:
    ${featureRequest}`,
  });

  // Workers: Execute the planned changes
  const fileChanges = await Promise.all(
    implementationPlan.files.map(async file => {
      // Each worker is specialized for the type of change
      const workerSystemPrompt = {
        create: 'You are an expert at implementing new files following best pract
        modify: 'You are an expert at modifying existing code while maintaining c
        delete: 'You are an expert at safely removing code while ensuring no brea
      }[file.changeType];

      const { object: change } = await generateObject({
        model: openai('gpt-4o'),
        schema: z.object({
          explanation: z.string(),
          code: z.string(),
        }),
        system: workerSystemPrompt,
        prompt: `Implement the changes for ${file.filePath} to support:
        ${file.purpose}

        Consider the overall feature context:
        ${featureRequest}`,
      });

      return {
        file,
        implementation: change,
      };
    }),
  );

  return {
    plan: implementationPlan,
    changes: fileChanges,
  };
}
```

Evaluator-Optimizer

This pattern introduces quality control into workflows by having dedicated evaluation steps that assess intermediate results. Based on the evaluation, the workflow can either proceed, retry with adjusted parameters, or take corrective action. This creates more robust workflows capable of self-improvement and error recovery.

```
import { openai } from '@ai-sdk/openai';

import { generateText, generateObject } from 'ai';

import { z } from 'zod';

async function translateWithFeedback(text: string, targetLanguage: string) {
  let currentTranslation = '';
  let iterations = 0;
  const MAX_ITERATIONS = 3;

  // Initial translation
  const { text: translation } = await generateText({
    model: openai('gpt-4o-mini'), // use small model for first attempt
    system: 'You are an expert literary translator.',
    prompt: `Translate this text to ${targetLanguage}, preserving tone and cultur
    ${text}`,
  });

  currentTranslation = translation;

  // Evaluation-optimization loop
  while (iterations < MAX_ITERATIONS) {
    // Evaluate current translation
    const { object: evaluation } = await generateObject({
      model: openai('gpt-4o'), // use a larger model to evaluate
      schema: z.object({
        qualityScore: z.number().min(1).max(10),
        preservesTone: z.boolean(),
        preservesNuance: z.boolean(),
        culturallyAccurate: z.boolean(),
        specificIssues: z.array(z.string()),
        improvementSuggestions: z.array(z.string()),
      }),
      system: 'You are an expert in evaluating literary translations.',
      prompt: `Evaluate this translation:

      Original: ${text}
      Translation: ${currentTranslation}

      Consider:
      1. Overall quality
      2. Preservation of tone
      3. Preservation of nuance
      4. Cultural accuracy`,
    });
```

```
    // Check if quality meets threshold
    if (
      evaluation.qualityScore >= 8 &&
      evaluation.preservesTone &&
      evaluation.preservesNuance &&
      evaluation.culturallyAccurate
    ) {
      break;
    }

    // Generate improved translation based on feedback
    const { text: improvedTranslation } = await generateText({
      model: openai('gpt-4o'), // use a larger model
      system: 'You are an expert literary translator.',
      prompt: `Improve this translation based on the following feedback:
      ${evaluation.specificIssues.join('\n')}
      ${evaluation.improvementSuggestions.join('\n')}

      Original: ${text}
      Current Translation: ${currentTranslation}`,
    });

    currentTranslation = improvedTranslation;
    iterations++;
  }

  return {
    finalTranslation: currentTranslation,
    iterationsRequired: iterations,
  };
}
```

Multi-Step Tool Usage

If your use case involves solving problems where the solution path is poorly defined or too complex to map out as a workflow in advance, you may want to provide the LLM with a set of lower-level tools and allow it to break down the task into small pieces that it can solve on its own iteratively, without discrete instructions. To implement this kind of agentic pattern, you need to call an LLM in a loop until a task is complete. The AI SDK makes this simple with the `stopWhen` parameter.

The AI SDK gives you control over the stopping conditions, enabling you to keep the LLM running until one of the conditions are met. The SDK automatically triggers an additional request to the model after every tool result (each request is considered a "step"), continuing until the model does not generate a tool call or other stopping conditions (e.g. `stepCountIs`) you define are satisfied.

`stopWhen` can be used with both `generateText` and `streamText`.

Using `stopWhen`

This example demonstrates how to create an agent that solves math problems.
It has a calculator tool (using math.js) that it can call to evaluate mathematical expressions.

```
import { openai } from '@ai-sdk/openai';
```

```
import { generateText, tool, stepCountIs } from 'ai';

import * as mathjs from 'mathjs';

import { z } from 'zod';

const { text: answer } = await generateText({
  model: openai('gpt-4o-2024-08-06'),
  tools: {
    calculate: tool({
      description:
        'A tool for evaluating mathematical expressions. ' +
        'Example expressions: ' +
        "'1.2 * (2 + 4.5)', '12.7 cm to inch', 'sin(45 deg) ^ 2'.",
      inputSchema: z.object({ expression: z.string() }),
      execute: async ({ expression }) => mathjs.evaluate(expression),
    }),
  },
  stopWhen: stepCountIs(10),
  system:
    'You are solving math problems. ' +
    'Reason step by step. ' +
    'Use the calculator when necessary. ' +
    'When you give the final answer, ' +
    'provide an explanation for how you arrived at it.',
  prompt:
    'A taxi driver earns $9461 per 1-hour of work. ' +
    'If he works 12 hours a day and in 1 hour ' +
    'he uses 12 liters of petrol with a price  of $134 for 1 liter. ' +
    'How much money does he earn in one day?',
});

console.log(`ANSWER: ${answer}`);
```

Structured Answers

When building an agent for tasks like mathematical analysis or report generation, it's often useful to have the agent's final output structured in a consistent format that your application can process. You can use an **answer tool** and the `toolChoice: 'required'` setting to force the LLM to answer with a structured output that matches the schema of the answer tool. The answer tool has no `execute` function, so invoking it will terminate the agent.

```
import { openai } from '@ai-sdk/openai';

import { generateText, tool, stepCountIs } from 'ai';

import 'dotenv/config';

import { z } from 'zod';

const { toolCalls } = await generateText({
  model: openai('gpt-4o-2024-08-06'),
  tools: {
    calculate: tool({
```

```
      description:
        'A tool for evaluating mathematical expressions. Example expressions: ' +
        "'1.2 * (2 + 4.5)', '12.7 cm to inch', 'sin(45 deg) ^ 2'.",
      inputSchema: z.object({ expression: z.string() }),
      execute: async ({ expression }) => mathjs.evaluate(expression),
    }),
    // answer tool: the LLM will provide a structured answer
    answer: tool({
      description: 'A tool for providing the final answer.',
      inputSchema: z.object({
        steps: z.array(
          z.object({
            calculation: z.string(),
            reasoning: z.string(),
          }),
        ),
        answer: z.string(),
      }),
      // no execute function - invoking it will terminate the agent
    }),
  },
  toolChoice: 'required',
  stopWhen: stepCountIs(10),
  system:
    'You are solving math problems. ' +
    'Reason step by step. ' +
    'Use the calculator when necessary. ' +
    'The calculator can only do simple additions, subtractions, multiplications,
    'When you give the final answer, provide an explanation for how you got it.',
  prompt:
    'A taxi driver earns $9461 per 1-hour work. ' +
    'If he works 12 hours a day and in 1 hour he uses 14-liters petrol with price
    'How much money does he earn in one day?',
});

console.log(`FINAL TOOL CALLS: ${JSON.stringify(toolCalls, null, 2)}`);
```

You can also use the `experimental_output` setting for `generateText` to generate structured outputs.

Accessing all steps

Calling `generateText` with `stopWhen` can result in several calls to the LLM (steps).
You can access information from all steps by using the `steps` property of the response.

```
import { generateText, stepCountIs } from 'ai';

const { steps } = await generateText({
  model: openai('gpt-4o'),
  stopWhen: stepCountIs(10),
  // ...
});
```

```
// extract all tool calls from the steps:
const allToolCalls = steps.flatMap(step => step.toolCalls);
```

Getting notified on each completed step

You can use the `onStepFinish` callback to get notified on each completed step.
It is triggered when a step is finished,
i.e. all text deltas, tool calls, and tool results for the step are available.

```
import { generateText, stepCountIs } from 'ai';

const result = await generateText({
  model: 'openai/gpt-4.1',
  stopWhen: stepCountIs(10),
  onStepFinish({ text, toolCalls, toolResults, finishReason, usage }) {
    // your own logic, e.g. for saving the chat history or recording usage
  },
  // ...
});
```

Previous — Next

On this page

Resources

Docs | Cookbook | Providers | Showcase | GitHub | Discussions

More

Playground | Contact Sales

About Vercel

Next.js + Vercel | Open Source Software | GitHub | X

Legal

Privacy Policy

Navigating the Library

The AI SDK is a powerful toolkit for building AI applications. This page will help you pick the right tools for your requirements.

Let's start with a quick overview of the AI SDK, which is comprised of three parts:

- **AI SDK Core**: A unified, provider agnostic API for generating text, structured objects, and tool calls with LLMs.
- **AI SDK UI**: A set of framework-agnostic hooks for building chat and generative user interfaces.
- AI SDK RSC: Stream generative user interfaces with React Server Components (RSC). Development is currently experimental and we recommend using AI SDK UI.

Choosing the Right Tool for Your Environment

When deciding which part of the AI SDK to use, your first consideration should be the environment and existing stack you are working with. Different components of the SDK are tailored to specific frameworks and environments.

Library	Purpose	Environment Compatibility
AI SDK Core	Call any LLM with unified API (e.g. generateText and generateObject)	Any JS environment (e.g. Node.js, Deno, Browser)
AI SDK UI	Build streaming chat and generative UIs (e.g. useChat)	React & Next.js, Vue & Nuxt, Svelte & SvelteKit
AI SDK RSC	Stream generative UIs from Server to Client (e.g. streamUI). Development is currently experimental and we recommend using AI SDK UI.	Any framework that supports React Server Components (e.g. Next.js)

Environment Compatibility

These tools have been designed to work seamlessly with each other and it's likely that you will be using them together. Let's look at how you could decide which libraries to use based on your application environment, existing stack, and requirements.

The following table outlines AI SDK compatibility based on environment:

Environment	AI SDK Core	AI SDK UI	AI SDK RSC
None / Node.js / Deno			
Vue / Nuxt			
Svelte / SvelteKit			
Next.js Pages Router			
Next.js App Router			

When to use AI SDK UI

AI SDK UI provides a set of framework-agnostic hooks for quickly building **production-ready AI-native applications**. It offers:

- Full support for streaming chat and client-side generative UI
- Utilities for handling common AI interaction patterns (i.e. chat, completion, assistant)

- Production-tested reliability and performance
- Compatibility across popular frameworks

AI SDK UI Framework Compatibility

AI SDK UI supports the following frameworks: React, Svelte, and Vue.js. Here is a comparison of the supported functions across these frameworks:

Function	React	Svelte	Vue.js
useChat			
useChat tool calling			
useCompletion			
useObject			

Contributions are welcome to implement missing features for non-React frameworks.

When to use AI SDK RSC

AI SDK RSC is currently experimental. We recommend using AI SDK UI for production. For guidance on migrating from RSC to UI, see our migration guide.

React Server Components (RSCs) provide a new approach to building React applications that allow components to render on the server, fetch data directly, and stream the results to the client, reducing bundle size and improving performance. They also introduce a new way to call server-side functions from anywhere in your application called Server Actions.

AI SDK RSC provides a number of utilities that allow you to stream values and UI directly from the server to the client. However, **it's important to be aware of current limitations**:

- **Cancellation**: currently, it is not possible to abort a stream using Server Actions. This will be improved in future releases of React and Next.js.
- **Increased Data Transfer**: using `createStreamableUI` can lead to quadratic data transfer (quadratic to the length of generated text). You can avoid this using `createStreamableValue` instead, and rendering the component client-side.
- **Re-mounting Issue During Streaming**: when using `createStreamableUI`, components re-mount on `.done()`, causing flickering.

Given these limitations, **we recommend using AI SDK UI for production applications**.

Previous: Getting Started | Next: Next.js App Router

Next.js App Router Quickstart

The AI SDK is a powerful Typescript library designed to help developers build AI-powered applications.

In this quickstart tutorial, you'll build a simple AI-chatbot with a streaming user interface. Along the way, you'll learn key concepts and techniques that are fundamental to using the SDK in your own projects.

If you are unfamiliar with the concepts of Prompt Engineering and HTTP Streaming, you can optionally read these documents first.

Prerequisites

To follow this quickstart, you'll need:

- Node.js 18+ and pnpm installed on your local development machine.
- An OpenAI API key.

If you haven't obtained your OpenAI API key, you can do so by signing up on the OpenAI website.

Create Your Application

Start by creating a new Next.js application. This command will create a new directory named `my-ai-app` and set up a basic Next.js application inside it.

Be sure to select yes when prompted to use the App Router and Tailwind CSS.
If you are looking for the Next.js Pages Router quickstart guide, you can find it here.

```
pnpm create next-app@latest my-ai-app
```

Navigate to the newly created directory:

```
cd my-ai-app
```

Install dependencies

Install `ai`, `@ai-sdk/react`, and `@ai-sdk/openai`, the AI package, AI SDK's React hooks, and AI SDK's OpenAI provider respectively.

The AI SDK is designed to be a unified interface to interact with any large language model. This means that you can change model and providers with just one line of code! Learn more about available providers and building custom providers in the providers section.

```
pnpm add ai @ai-sdk/react @ai-sdk/openai zod
```

Configure OpenAI API key

Create a `.env.local` file in your project root and add your OpenAI API Key. This key is used to authenticate your application with the OpenAI service.

```
touch .env.local
```

Edit the `.env.local` file:

```
OPENAI_API_KEY=xxxxxxxxx
```

Replace `xxxxxxxxx` with your actual OpenAI API key.

The AI SDK's OpenAI Provider will default to using the `OPENAI_API_KEY` environment variable.

Create a Route Handler

Create a route handler, `app/api/chat/route.ts` and add the following code:

```
import { openai } from '@ai-sdk/openai';
import { streamText, UIMessage, convertToModelMessages } from 'ai';

// Allow streaming responses up to 30 seconds
export const maxDuration = 30;

export async function POST(req: Request) {
  const { messages }: { messages: UIMessage[] } = await req.json();

  const result = streamText({
    model: openai('gpt-4o'),
    messages: convertToModelMessages(messages),
  });

  return result.toUIMessageStreamResponse();
}
```

Let's take a look at what is happening in this code:

1. Define an asynchronous `POST` request handler and extract `messages` from the body of the request. The `messages` variable contains a history of the conversation between you and the chatbot and provides the chatbot with the necessary context to make the next generation. The `messages` are of UIMessage type, which are designed for use in application UI - they contain the entire message history and associated metadata like timestamps.
2. Call `streamText`, which is imported from the `ai` package. This function accepts a configuration object that contains a `model` provider (imported from `@ai-sdk/openai`) and `messages` (defined in step 1). You can pass additional settings to further customise the model's behaviour. The `messages` key expects a `ModelMessage[]` array. This type is different from `UIMessage` in that it does not include metadata, such as timestamps or sender information. To convert between these types, we use the `convertToModelMessages` function, which strips the UI-specific metadata and transforms the `UIMessage[]` array into the `ModelMessage[]` format that the model expects.
3. The `streamText` function returns a `StreamTextResult`. This result object contains the toUIMessageStreamResponse function which converts the result to a streamed response object.
4. Finally, return the result to the client to stream the response.

This Route Handler creates a POST request endpoint at `/api/chat`.

Wire up the UI

Now that you have a Route Handler that can query an LLM, it's time to setup your frontend. The AI SDK's UI package abstracts the complexity of a chat interface into one hook, useChat.

Update your root page (`app/page.tsx`) with the following code to show a list of chat messages and provide a user message input:

```
'use client';

import { useChat } from '@ai-sdk/react';
import { useState } from 'react';

export default function Chat() {
  const [input, setInput] = useState('');
  const { messages, sendMessage } = useChat();

  return (
    <div className="flex flex-col w-full max-w-md py-24 mx-auto stretch">
      {messages.map((message) => (
        <div key={message.id} className="whitespace-pre-wrap">
          {message.role === 'user' ? 'User: ' : 'AI: '}
          {message.parts.map((part, i) => {
            switch (part.type) {
              case 'text':
                return (
                  <div key={`${message.id}-${i}`}>
                    {part.text}
                  </div>
                );
            }
          })}
        </div>
      ))}

      <form
        onSubmit={e => {
          e.preventDefault();
          sendMessage({ text: input });
          setInput('');
        }}
      >
        <input
          className="fixed dark:bg-zinc-900 bottom-0 w-full max-w-md p-2 mb-8 bor
          value={input}
          placeholder="Say something..."
          onChange={e => setInput(e.currentTarget.value)}
        />
      </form>
    </div>
  );
}
```

Make sure you add the `"use client"` directive to the top of your file. This allows you to add interactivity with Javascript.

This page utilizes the `useChat` hook, which will, by default, use the `POST` API route you created earlier (`/api/chat`). The hook provides functions and state for handling user input and form submission. The `useChat` hook provides multiple utility functions and state variables:

- `messages` - the current chat messages (an array of objects with `id`, `role`, and `parts` properties).
- `sendMessage` - a function to send a message to the chat API.

The component uses local state (`useState`) to manage the input field value, and handles form submission by calling `sendMessage` with the input text and then clearing the input field.

The LLM's response is accessed through the message `parts` array. Each message contains an ordered array of `parts` that represents everything the model generated in its response. These parts can include plain text, reasoning tokens, and more that you will see later. The `parts` array preserves the sequence of the model's outputs, allowing you to display or process each component in the order it was generated.

Running Your Application

With that, you have built everything you need for your chatbot! To start your application, use the command:

```
pnpm run dev
```

Head to your browser and open http://localhost:3000. You should see an input field. Test it out by entering a message and see the AI chatbot respond in real-time! The AI SDK makes it fast and easy to build AI chat interfaces with Next.js.

Enhance Your Chatbot with Tools

While large language models (LLMs) have incredible generation capabilities, they struggle with discrete tasks (e.g. mathematics) and interacting with the outside world (e.g. getting the weather). This is where tools come in.

Tools are actions that an LLM can invoke. The results of these actions can be reported back to the LLM to be considered in the next response.

For example, if a user asks about the current weather, without tools, the model would only be able to provide general information based on its training data. But with a weather tool, it can fetch and provide up-to-date, location-specific weather information.

Let's enhance your chatbot by adding a simple weather tool.

Update Your Route Handler

Modify your `app/api/chat/route.ts` file to include the new weather tool:

```
import { openai } from '@ai-sdk/openai';
import { streamText, UIMessage, convertToModelMessages, tool } from 'ai';
import { z } from 'zod';

export const maxDuration = 30;

export async function POST(req: Request) {
  const { messages }: { messages: UIMessage[] } = await req.json();

  const result = streamText({
    model: openai('gpt-4o'),
    messages: convertToModelMessages(messages),
```

```
    tools: {
      weather: tool({
        description: 'Get the weather in a location (fahrenheit)',
        inputSchema: z.object({
          location: z.string().describe('The location to get the weather for'),
        }),
        execute: async ({ location }) => {
          const temperature = Math.round(Math.random() * (90 - 32) + 32);
          return {
            location,
            temperature,
          };
        },
      }),
    },
  });

  return result.toUIMessageStreamResponse();
}
```

In this updated code:

1. You import the `tool` function from the `ai` package and `z` from `zod` for schema validation.
2. You define a `tools` object with a `weather` tool. This tool:
   - Has a description that helps the model understand when to use it.
   - Defines `inputSchema` using a Zod schema, specifying that it requires a `location` string to execute this tool. The model will attempt to extract this input from the context of the conversation. If it can't, it will ask the user for the missing information.
   - Defines an `execute` function that simulates getting weather data (in this case, it returns a random temperature). This is an asynchronous function running on the server so you can fetch real data from an external API.

Now your chatbot can "fetch" weather information for any location the user asks about. When the model determines it needs to use the weather tool, it will generate a tool call with the necessary input. The `execute` function will then be automatically run, and the tool output will be added to the `messages` as a `tool` message.

Try asking something like "What's the weather in New York?" and see how the model uses the new tool.

Notice the blank response in the UI? This is because instead of generating a text response, the model generated a tool call. You can access the tool call and subsequent tool result on the client via the `tool-weather` part of the `message.parts` array.

Tool parts are always named `tool-{toolName}`, where `{toolName}` is the key you used when defining the tool. In this case, since we defined the tool as `weather`, the part type is `tool-weather`.

Update the UI

To display the tool invocation in your UI, update your `app/page.tsx` file:

```
'use client';

import { useChat } from '@ai-sdk/react';
import { useState } from 'react';
```

```
export default function Chat() {
  const [input, setInput] = useState('');
  const { messages, sendMessage } = useChat();

  return (
    <div classNam="flex flex-col w-full max-w-md py-24 mx-auto stretch">
      {messages.map((message) => (
        <div key={message.id} classNam="whitespace-pre-wrap">
          {message.role === 'user' ? 'User: ' : 'AI: '}
          {message.parts.map((part, i) => {
            switch (part.type) {
              case 'text':
                return (
                  <div key={`${message.id}-${i}`}>
                    {part.text}
                  </div>
                );
              case 'tool-weather':
                return (
                  <pre key={`${message.id}-${i}`}>
                    {JSON.stringify(part, null, 2)}
                  </pre>
                );
            }
          })}
        </div>
      ))}

      <form
        onSubmit={e => {
          e.preventDefault();
          sendMessage({ text: input });
          setInput('');
        }}
      >
        <input
          classNam="fixed dark:bg-zinc-900 bottom-0 w-full max-w-md p-2 mb-8 bor
          value={input}
          placeholder="Say something..."
          onChange={e => setInput(e.currentTarget.value)}
        />
      </form>
    </div>
  );
}
```

With this change, you're updating the UI to handle different message parts. For text parts, you display the text content as before. For weather tool invocations, you display a JSON representation of the tool call and its result.

Now, when you ask about the weather, you'll see the tool call and its result displayed in your chat interface.

Enabling Multi-Step Tool Calls

You may have noticed that while the tool is now visible in the chat interface, the model isn't using this information to answer your original query. This is because once the model generates a tool call, it has technically completed its generation.

To solve this, you can enable multi-step tool calls using `stopWhen`. By default, `stopWhen` is set to `stepCountIs(1)`, which means generation stops after the first step when there are tool results. By changing this condition, you can allow the model to automatically send tool results back to itself to trigger additional generations until your specified stopping condition is met. In this case, you want the model to continue generating so it can use the weather tool results to answer your original question.

Update Your Route Handler

Modify your `app/api/chat/route.ts` file to include the `stopWhen` condition:

```
import { openai } from '@ai-sdk/openai';
import {
  streamText,
  UIMessage,
  convertToModelMessages,
  tool,
  stepCountIs,
} from 'ai';
import { z } from 'zod';

export const maxDuration = 30;

export async function POST(req: Request) {
  const { messages }: { messages: UIMessage[] } = await req.json();

  const result = streamText({
    model: openai('gpt-4o'),
    messages: convertToModelMessages(messages),
    stopWhen: stepCountIs(5),
    tools: {
      weather: tool({
        description: 'Get the weather in a location (fahrenheit)',
        inputSchema: z.object({
          location: z.string().describe('The location to get the weather for'),
        }),
        execute: async ({ location }) => {
          const temperature = Math.round(Math.random() * (90 - 32) + 32);
          return {
            location,
            temperature,
          };
        },
      }),
    },
  });

  return result.toUIMessageStreamResponse();
}
```

In this updated code:

1. You set `stopWhen` to be when `stepCountIs` 5, allowing the model to use up to 5 "steps" for any given generation.
2. You add an `onStepFinish` callback to log any `toolResults` from each step of the interaction, helping you understand the model's tool usage. This means we can also delete the `toolCall` and `toolResult` `console.log` statements from the previous example.

Head back to the browser and ask about the weather in a location. You should now see the model using the weather tool results to answer your question.

By setting `stopWhen: stepCountIs(5)`, you're allowing the model to use up to 5 "steps" for any given generation. This enables more complex interactions and allows the model to gather and process information over several steps if needed. You can see this in action by adding another tool to convert the temperature from Celsius to Fahrenheit.

Add another tool

Update your `app/api/chat/route.ts` file to add a new tool to convert the temperature from Fahrenheit to Celsius:

```
import { openai } from '@ai-sdk/openai';
import {
  streamText,
  UIMessage,
  convertToModelMessages,
  tool,
  stepCountIs,
} from 'ai';
import { z } from 'zod';

export const maxDuration = 30;

export async function POST(req: Request) {
  const { messages }: { messages: UIMessage[] } = await req.json();

  const result = streamText({
    model: openai('gpt-4o'),
    messages: convertToModelMessages(messages),
    stopWhen: stepCountIs(5),
    tools: {
      weather: tool({
        description: 'Get the weather in a location (fahrenheit)',
        inputSchema: z.object({
          location: z.string().describe('The location to get the weather for'),
        }),
        execute: async ({ location }) => {
          const temperature = Math.round(Math.random() * (90 - 32) + 32);
          return {
            location,
            temperature,
          };
        },
      }),
      convertFahrenheitToCelsius: tool({
```

```
        description: 'Convert a temperature in fahrenheit to celsius',
        inputSchema: z.object({
          temperature: z.number().describe('The temperature in fahrenheit to conv
        }),
        execute: async ({ temperature }) => {
          const celsius = Math.round((temperature - 32) * (5 / 9));
          return {
            celsius,
          };
        },
      }),
    },
  });

  return result.toUIMessageStreamResponse();
}
```

Update Your Frontend

Update your `app/page.tsx` file to render the new temperature conversion tool:

```
'use client';

import { useChat } from '@ai-sdk/react';
import { useState } from 'react';

export default function Chat() {
  const [input, setInput] = useState('');
  const { messages, sendMessage } = useChat();

  return (
    <div className="flex flex-col w-full max-w-md py-24 mx-auto stretch">
      {messages.map((message) => (
        <div key={message.id} className="whitespace-pre-wrap">
          {message.role === 'user' ? 'User: ' : 'AI: '}
          {message.parts.map((part, i) => {
            switch (part.type) {
              case 'text':
                return (
                  <div key={`${message.id}-${i}`}>
                    {part.text}
                  </div>
                );
              case 'tool-weather':
              case 'tool-convertFahrenheitToCelsius':
                return (
                  <pre key={`${message.id}-${i}`}>
                    {JSON.stringify(part, null, 2)}
                  </pre>
                );
            }
          })}
        </div>
      ))}
```

```
      <form
        onSubmit={e => {
          e.preventDefault();
          sendMessage({ text: input });
          setInput('');
        }}
      >
        <input
          className="fixed dark:bg-zinc-900 bottom-0 w-full max-w-md p-2 mb-8 bor
          value={input}
          placeholder="Say something..."
          onChange={e => setInput(e.currentTarget.value)}
        />
      </form>
    </div>
  );
}
```

This update handles the new `tool-convertFahrenheitToCelsius` part type, displaying the temperature conversion tool calls and results in the UI.

Now, when you ask "What's the weather in New York in celsius?", you should see a more complete interaction:

1. The model will call the weather tool for New York.
2. You'll see the tool output displayed.
3. It will then call the temperature conversion tool to convert the temperature from Fahrenheit to Celsius.
4. The model will then use that information to provide a natural language response about the weather in New York.

This multi-step approach allows the model to gather information and use it to provide more accurate and contextual responses, making your chatbot considerably more useful.

This simple example demonstrates how tools can expand your model's capabilities. You can create more complex tools to integrate with real APIs, databases, or any other external systems, allowing the model to access and process real-world data in real-time. Tools bridge the gap between the model's knowledge cutoff and current information.

Where to Next?

You've built an AI chatbot using the AI SDK! From here, you have several paths to explore:

- To learn more about the AI SDK, read through the documentation.
- If you're interested in diving deeper with guides, check out the RAG (retrieval-augmented generation) and multi-modal chatbot guides.
- To jumpstart your first AI project, explore available templates.

Previous: Navigating the Library | Next: Next.js Pages Router

Next.js Pages Router Quickstart

The AI SDK is a powerful Typescript library designed to help developers build AI-powered applications.

In this quickstart tutorial, you'll build a simple AI-chatbot with a streaming user interface. Along the way, you'll learn key concepts and techniques that are fundamental to using the SDK in your own projects.

If you are unfamiliar with the concepts of Prompt Engineering and HTTP Streaming, you can optionally read these documents first.

Prerequisites

To follow this quickstart, you'll need:

- Node.js 18+ and pnpm installed on your local development machine.
- An OpenAI API key.

If you haven't obtained your OpenAI API key, you can do so by signing up on the OpenAI website.

Setup Your Application

Start by creating a new Next.js application. This command will create a new directory named `my-ai-app` and set up a basic Next.js application inside it.

Be sure to select no when prompted to use the App Router. If you are looking for the Next.js App Router quickstart guide, you can find it here.

```
pnpm create next-app@latest my-ai-app
```

Navigate to the newly created directory:

```
cd my-ai-app
```

Install dependencies

Install `ai`, `@ai-sdk/react`, and `@ai-sdk/openai`, the AI package, AI SDK's React hooks, and AI SDK's OpenAI provider respectively.

The AI SDK is designed to be a unified interface to interact with any large language model. This means that you can change model and providers with just one line of code! Learn more about available providers and building custom providers in the providers section.

```
pnpm add ai @ai-sdk/react @ai-sdk/openai zod
```

Configure OpenAI API key

Create a `.env.local` file in your project root and add your OpenAI API Key. This key is used to authenticate your application with the OpenAI service.

```
touch .env.local
```

Edit the `.env.local` file:

```
OPENAI_API_KEY=xxxxxxxxx
```

Replace `xxxxxxxxx` with your actual OpenAI API key.

The AI SDK's OpenAI Provider will default to using the `OPENAI_API_KEY` environment variable.

Create a Route Handler

As long as you are on Next.js 13+, you can use Route Handlers (using the App Router) alongside the Pages Router. This is recommended to enable you to use the Web APIs interface/signature and to better support streaming.

Create a Route Handler (`app/api/chat/route.ts`) and add the following code:

```
import { openai } from '@ai-sdk/openai';

import { streamText, UIMessage, convertToModelMessages } from 'ai';

// Allow streaming responses up to 30 seconds
export const maxDuration = 30;

export async function POST(req: Request) {
  const { messages }: { messages: UIMessage[] } = await req.json();

  const result = streamText({
    model: openai('gpt-4o'),
    messages: convertToModelMessages(messages),
  });

  return result.toUIMessageStreamResponse();
}
```

Let's take a look at what is happening in this code:

1. Define an asynchronous `POST` request handler and extract `messages` from the body of the request. The `messages` variable contains a history of the conversation between you and the chatbot and provides the chatbot with the necessary context to make the next generation. The `messages` are of UIMessage type, which are designed for use in application UI - they contain the entire message history and associated metadata like timestamps.

2. Call `streamText` , which is imported from the `ai` package. This function accepts a configuration object that contains a `model` provider (imported from `@ai-sdk/openai`) and `messages` (defined in step 1). You can pass additional settings to further customise the model's behaviour. The `messages` key expects a `ModelMessage[]` array. This type is different from `UIMessage` in that it does not include metadata, such as timestamps or sender information. To convert between these types, we use the `convertToModelMessages` function, which strips the UI-specific metadata and transforms the `UIMessage[]` array into the `ModelMessage[]` format that the model expects.

3. The `streamText` function returns a `StreamTextResult` . This result object contains the `toUIMessageStreamResponse` function which converts the result to a streamed response object.

4. Finally, return the result to the client to stream the response.

This Route Handler creates a POST request endpoint at `/api/chat` .

Wire up the UI

Now that you have an API route that can query an LLM, it's time to setup your frontend. The AI SDK's UI package abstracts the complexity of a chat interface into one hook, useChat.

Update your root page (`pages/index.tsx`) with the following code to show a list of chat messages and provide a user message input:

```
import { useChat } from '@ai-sdk/react';
import { useState } from 'react';

export default function Chat() {
  const [input, setInput] = useState('');
  const { messages, sendMessage } = useChat();

  return (
    <div classNam="flex flex-col w-full max-w-md py-24 mx-auto stretch">
      {messages.map((message) => (
        <div key={message.id} className="whitespace-pre-wrap">
          {message.role === 'user' ? 'User: ' : 'AI: '}
          {message.parts.map((part, i) => {
            switch (part.type) {
              case 'text':
                return <div key={`${message.id}-${i}`}>{part.text}</div>;
            }
          })}
        </div>
      ))}

      <form
        onSubmit={(e) => {
          e.preventDefault();
          sendMessage({ text: input });
          setInput('');
        }}
      >
        <input
          className="fixed dark:bg-zinc-900 bottom-0 w-full max-w-md p-2 mb-8 bor
          value={input}
          placeholder="Say something..."
          onChange={(e) => setInput(e.currentTarget.value)}
        />
      </form>
    </div>
  );
}
```

This page utilizes the `useChat` hook, which will, by default, use the `POST` API route you created earlier (`/api/chat`). The hook provides functions and state for handling user input and form submission. The `useChat` hook provides multiple utility functions and state variables:

- `messages` - the current chat messages (an array of objects with `id` , `role` , and `parts` properties).
- `sendMessage` - a function to send a message to the chat API.

The component uses local state (`useState`) to manage the input field value, and handles form submission by calling `sendMessage` with the input text and then clearing the input field.

The LLM's response is accessed through the message `parts` array. Each message contains an ordered array of `parts` that represents everything the model generated in its response. These parts can include plain text, reasoning tokens, and more that you will see later. The `parts` array preserves the sequence of the model's outputs, allowing you to display or process each component in the order it was generated.

Running Your Application

With that, you have built everything you need for your chatbot! To start your application, use the command:

```
pnpm run dev
```

Head to your browser and open http://localhost:3000. You should see an input field. Test it out by entering a message and see the AI chatbot respond in real-time! The AI SDK makes it fast and easy to build AI chat interfaces with Next.js.

Enhance Your Chatbot with Tools

While large language models (LLMs) have incredible generation capabilities, they struggle with discrete tasks (e.g. mathematics) and interacting with the outside world (e.g. getting the weather). This is where tools come in.

Tools are actions that an LLM can invoke. The results of these actions can be reported back to the LLM to be considered in the next response.

For example, if a user asks about the current weather, without tools, the model would only be able to provide general information based on its training data. But with a weather tool, it can fetch and provide up-to-date, location-specific weather information.

Update Your Route Handler

Let's start by giving your chatbot a weather tool. Update your Route Handler (`app/api/chat/route.ts`):

```
import { openai } from '@ai-sdk/openai';

import { streamText, UIMessage, convertToModelMessages, tool } from 'ai';

import { z } from 'zod';

// Allow streaming responses up to 30 seconds
export const maxDuration = 30;

export async function POST(req: Request) {
  const { messages }: { messages: UIMessage[] } = await req.json();

  const result = streamText({
    model: openai('gpt-4o'),
    messages: convertToModelMessages(messages),
    tools: {
```

```
      weather: tool({
        description: 'Get the weather in a location',
        inputSchema: z.object({
          location: z.string().describe('The location to get the weather for'),
        }),
        execute: async ({ location }) => ({
          location,
          temperature: 72 + Math.floor(Math.random() * 21) - 10,
        }),
      }),
    },
  });

  return result.toUIMessageStreamResponse();
}
```

In this updated code:

1. You import the `tool` function from the `ai` package and `z` from `zod` for schema validation.
2. You define a `tools` object with a `weather` tool. This tool:
   - Has a description that helps the model understand when to use it.
   - Defines `inputSchema` using a Zod schema, specifying that it requires a `location` string to execute this tool. The model will attempt to extract this input from the context of the conversation. If it can't, it will ask the user for the missing information.
   - Defines an `execute` function that simulates getting weather data (in this case, it returns a random temperature). This is an asynchronous function running on the server so you can fetch real data from an external API.

Now your chatbot can "fetch" weather information for any location the user asks about. When the model determines it needs to use the weather tool, it will generate a tool call with the necessary input. The `execute` function will then be automatically run, and the tool output will be added to the `messages` as a `tool` message.

Try asking something like "What's the weather in New York?" and see how the model uses the new tool.

Notice the blank response in the UI? This is because instead of generating a text response, the model generated a tool call. You can access the tool call and subsequent tool result on the client via the `tool-weather` part of the `message.parts` array.

Tool parts are always named `tool-{toolName}`, where `{toolName}` is the key you used when defining the tool. In this case, since we defined the tool as `weather`, the part type is `tool-weather`.

Update the UI

To display the tool invocations in your UI, update your `pages/index.tsx` file:

```
import { useChat } from '@ai-sdk/react';
import { useState } from 'react';

export default function Chat() {
  const [input, setInput] = useState('');
  const { messages, sendMessage } = useChat();
```

```
  return (
    <div className="flex flex-col w-full max-w-md py-24 mx-auto stretch">
      {messages.map((message) => (
        <div key={message.id} className="whitespace-pre-wrap">
          {message.role === 'user' ? 'User: ' : 'AI: '}
          {message.parts.map((part, i) => {
            switch (part.type) {
              case 'text':
                return <div key={`${message.id}-${i}`}>{part.text}</div>;
              case 'tool-weather':
                return (
                  <pre key={`${message.id}-${i}`}>
                    {JSON.stringify(part, null, 2)}
                  </pre>
                );
            }
          })}
        </div>
      ))}

      <form
        onSubmit={(e) => {
          e.preventDefault();
          sendMessage({ text: input });
          setInput('');
        }}
      >
        <input
          className="fixed dark:bg-zinc-900 bottom-0 w-full max-w-md p-2 mb-8 bor
          value={input}
          placeholder="Say something..."
          onChange={(e) => setInput(e.currentTarget.value)}
        />
      </form>
    </div>
  );
}
```

With this change, you're updating the UI to handle different message parts. For text parts, you display the text content as before. For weather tool invocations, you display a JSON representation of the tool call and its result.

Now, when you ask about the weather, you'll see the tool call and its result displayed in your chat interface.

Enabling Multi-Step Tool Calls

You may have noticed that while the tool is now visible in the chat interface, the model isn't using this information to answer your original query. This is because once the model generates a tool call, it has technically completed its generation.

To solve this, you can enable multi-step tool calls using `stopWhen`. By default, `stopWhen` is set to `stepCountIs(1)`, which means generation stops after the first step when there are tool results. By changing this condition, you can allow the model to automatically send tool results back to itself to

trigger additional generations until your specified stopping condition is met. In this case, you want the model to continue generating so it can use the weather tool results to answer your original question.

Update Your Route Handler

Modify your `app/api/chat/route.ts` file to include the `stopWhen` condition:

```
import { openai } from '@ai-sdk/openai';

import {
  streamText,
  UIMessage,
  convertToModelMessages,
  tool,
  stepCountIs,
} from 'ai';

import { z } from 'zod';

export const maxDuration = 30;

export async function POST(req: Request) {
  const { messages }: { messages: UIMessage[] } = await req.json();

  const result = streamText({
    model: openai('gpt-4o'),
    messages: convertToModelMessages(messages),
    stopWhen: stepCountIs(5),
    tools: {
      weather: tool({
        description: 'Get the weather in a location (fahrenheit)',
        inputSchema: z.object({
          location: z.string().describe('The location to get the weather for'),
        }),
        execute: async ({ location }) => {
          const temperature = Math.round(Math.random() * (90 - 32) + 32);
          return {
            location,
            temperature,
          };
        },
      }),
    },
  });

  return result.toUIMessageStreamResponse();
}
```

Head back to the browser and ask about the weather in a location. You should now see the model using the weather tool results to answer your question.

By setting `stopWhen: stepCountIs(5)`, you're allowing the model to use up to 5 "steps" for any given generation. This enables more complex interactions and allows the model to gather and process information over several steps if needed. You can see this in action by adding another tool to convert the temperature from Celsius to Fahrenheit.

Add another tool

Update your `app/api/chat/route.ts` file to add a new tool to convert the temperature from Fahrenheit to Celsius:

```
import { openai } from '@ai-sdk/openai';

import {
  streamText,
  UIMessage,
  convertToModelMessages,
  tool,
  stepCountIs,
} from 'ai';

import { z } from 'zod';

export const maxDuration = 30;

export async function POST(req: Request) {
  const { messages }: { messages: UIMessage[] } = await req.json();

  const result = streamText({
    model: openai('gpt-4o'),
    messages: convertToModelMessages(messages),
    stopWhen: stepCountIs(5),
    tools: {
      weather: tool({
        description: 'Get the weather in a location (fahrenheit)',
        inputSchema: z.object({
          location: z.string().describe('The location to get the weather for'),
        }),
        execute: async ({ location }) => {
          const temperature = Math.round(Math.random() * (90 - 32) + 32);
          return {
            location,
            temperature,
          };
        },
      }),
      convertFahrenheitToCelsius: tool({
        description: 'Convert a temperature in fahrenheit to celsius',
        inputSchema: z.object({
          temperature: z.number().describe('The temperature in fahrenheit to conv
        }),
        execute: async ({ temperature }) => {
          const celsius = Math.round((temperature - 32) * (5 / 9));
          return {
            celsius,
          };
        },
      }),
    },
  });
```

```
  return result.toUIMessageStreamResponse();
}
```

Update Your Frontend

Update your `pages/index.tsx` file to render the new temperature conversion tool:

```
import { useChat } from '@ai-sdk/react';
import { useState } from 'react';

export default function Chat() {
  const [input, setInput] = useState('');
  const { messages, sendMessage } = useChat();

  return (
    <div classNam="flex flex-col w-full max-w-md py-24 mx-auto stretch">
      {messages.map((message) => (
        <div key={message.id} className="whitespace-pre-wrap">
          {message.role === 'user' ? 'User: ' : 'AI: '}
          {message.parts.map((part, i) => {
            switch (part.type) {
              case 'text':
                return <div key={`${message.id}-${i}`}>{part.text}</div>;
              case 'tool-weather':
              case 'tool-convertFahrenheitToCelsius':
                return (
                  <pre key={`${message.id}-${i}`}>
                    {JSON.stringify(part, null, 2)}
                  </pre>
                );
            }
          })}
        </div>
      ))}

      <form
        onSubmit={(e) => {
          e.preventDefault();
          sendMessage({ text: input });
          setInput('');
        }}
      >
        <input
          className="fixed dark:bg-zinc-900 bottom-0 w-full max-w-md p-2 mb-8 bor
          value={input}
          placeholder="Say something..."
          onChange={(e) => setInput(e.currentTarget.value)}
        />
      </form>
    </div>
  );
}
```

This update handles the new `tool-convertFahrenheitToCelsius` part type, displaying the temperature conversion tool calls and results in the UI.

Now, when you ask "What's the weather in New York in celsius?", you should see a more complete interaction:

1. The model will call the weather tool for New York.
2. You'll see the tool output displayed.
3. It will then call the temperature conversion tool to convert the temperature from Fahrenheit to Celsius.
4. The model will then use that information to provide a natural language response about the weather in New York.

This multi-step approach allows the model to gather information and use it to provide more accurate and contextual responses, making your chatbot considerably more useful.

This simple example demonstrates how tools can expand your model's capabilities. You can create more complex tools to integrate with real APIs, databases, or any other external systems, allowing the model to access and process real-world data in real-time. Tools bridge the gap between the model's knowledge cutoff and current information.

Where to Next?

You've built an AI chatbot using the AI SDK! From here, you have several paths to explore:

- To learn more about the AI SDK, read through the documentation.
- If you're interested in diving deeper with guides, check out the RAG (retrieval-augmented generation) and multi-modal chatbot guides.
- To jumpstart your first AI project, explore available templates.

Svelte Quickstart

The AI SDK is a powerful Typescript library designed to help developers build AI-powered applications.

In this quickstart tutorial, you'll build a simple AI-chatbot with a streaming user interface. Along the way, you'll learn key concepts and techniques that are fundamental to using the SDK in your own projects.

If you are unfamiliar with the concepts of Prompt Engineering and HTTP Streaming, you can optionally read these documents first.

Prerequisites

To follow this quickstart, you'll need:

- Node.js 18+ and pnpm installed on your local development machine.
- An OpenAI API key.

If you haven't obtained your OpenAI API key, you can do so by signing up on the OpenAI website.

Set Up Your Application

This guide applies to SvelteKit versions 4 and below.

Start by creating a new SvelteKit application. This command will create a new directory named `my-ai-app` and set up a basic SvelteKit application inside it.

```
npx sv create my-ai-app
```

Navigate to the newly created directory:

```
cd my-ai-app
```

Install Dependencies

Install `ai` and `@ai-sdk/openai`, the AI SDK's OpenAI provider.

> The AI SDK is designed to be a unified interface to interact with any large language model. This means that you can change model and providers with just one line of code! Learn more about available providers and building custom providers in the providers section.

Choose your package manager:

```
pnpm add -D ai @ai-sdk/openai @ai-sdk/svelte zod
# or
npm install -D ai @ai-sdk/openai @ai-sdk/svelte zod
# or
yarn add -D ai @ai-sdk/openai @ai-sdk/svelte zod
# or
bun add -d ai @ai-sdk/openai @ai-sdk/svelte zod
```

Configure OpenAI API Key

Create a `.env.local` file in your project root and add your OpenAI API Key. This key is used to authenticate your application with the OpenAI service.

```
touch .env.local
```

Edit the `.env.local` file:

```
OPENAI_API_KEY=xxxxxxxxx
```

Replace `xxxxxxxxx` with your actual OpenAI API key.

> Vite does not automatically load environment variables onto `process.env`, so you'll need to import `OPENAI_API_KEY` from `$env/static/private` in your code (see below).

Create an API route

Create a SvelteKit Endpoint, `src/routes/api/chat/+server.ts` and add the following code:

```
import { createOpenAI } from '@ai-sdk/openai';

import { streamText, type UIMessage, convertToModelMessages } from 'ai';

import { OPENAI_API_KEY } from '$env/static/private';

const openai = createOpenAI({
  apiKey: OPENAI_API_KEY,
});

export async function POST({ request }) {
  const { messages }: { messages: UIMessage[] } = await request.json();

  const result = streamText({
    model: openai('gpt-4o'),
    messages: convertToModelMessages(messages),
  });

  return result.toUIMessageStreamResponse();
}
```

If you see type errors with `OPENAI_API_KEY` or your `POST` function, run the dev server.

Let's take a look at what is happening in this code:

1. Create an OpenAI provider instance with the `createOpenAI` function from the `@ai-sdk/openai` package.
2. Define a `POST` request handler and extract `messages` from the body of the request. The `messages` variable contains a history of the conversation between you and the chatbot and provides the chatbot with the necessary context to make the next generation. The `messages` are of `UIMessage` type, which are designed for use in application UI - they contain the entire message history and associated metadata like timestamps.

3. Call `streamText`, which is imported from the `ai` package. This function accepts a configuration object that contains a `model` provider (defined in step 1) and `messages` (defined in step 2). You can pass additional settings to further customise the model's behaviour. The `messages` key expects a `ModelMessage[]` array. This type is different from `UIMessage` in that it does not include metadata, such as timestamps or sender information. To convert between these types, we use the `convertToModelMessages` function, which strips the UI-specific metadata and transforms the `UIMessage[]` array into the `ModelMessage[]` format that the model expects.
4. The `streamText` function returns a `StreamTextResult`. This result object contains the `toUIMessageStreamResponse` function which converts the result to a streamed response object.
5. Return the result to the client to stream the response.

Wire up the UI

Now that you have an API route that can query an LLM, it's time to set up your frontend. The AI SDK's UI package abstracts the complexity of a chat interface into one class, `Chat`.
Its properties and API are largely the same as React's `useChat`.

Update your root page (`src/routes/+page.svelte`) with the following code to show a list of chat messages and provide a user message input:

```
<script lang="ts">
  import { Chat } from '@ai-sdk/svelte';

  let input = '';
  const chat = new Chat({});

  function handleSubmit(event: SubmitEvent) {
    event.preventDefault();
    chat.sendMessage({ text: input });
    input = '';
  }
</script>

<main>
  <ul>
    {#each chat.messages as message, messageIndex (messageIndex)}
      <li>
        <div>{message.role}</div>
        <div>
          {#each message.parts as part, partIndex (partIndex)}
            {#if part.type === 'text'}
              <div>{part.text}</div>
            {/if}
          {/each}
        </div>
      </li>
    {/each}
  </ul>
  <form onsubmit={handleSubmit}>
    <input bind:value={input} />
    <button type="submit">Send</button>
```

```
  </form>
</main>
```

This page utilizes the `Chat` class, which will, by default, use the `POST` route handler you created earlier. The class provides functions and state for handling user input and form submission. The `Chat` class provides multiple utility functions and state variables:

- `messages` - the current chat messages (an array of objects with `id`, `role`, and `parts` properties).
- `sendMessage` - a function to send a message to the chat API.

The component uses local state to manage the input field value, and handles form submission by calling `sendMessage` with the input text and then clearing the input field.

The LLM's response is accessed through the message `parts` array. Each message contains an ordered array of `parts` that represents everything the model generated in its response. These parts can include plain text, reasoning tokens, and more that you will see later. The `parts` array preserves the sequence of the model's outputs, allowing you to display or process each component in the order it was generated.

Running Your Application

With that, you have built everything you need for your chatbot! To start your application, use the command:

```
pnpm run dev
```

Head to your browser and open http://localhost:5173. You should see an input field. Test it out by entering a message and see the AI chatbot respond in real-time! The AI SDK makes it fast and easy to build AI chat interfaces with Svelte.

Enhance Your Chatbot with Tools

While large language models (LLMs) have incredible generation capabilities, they struggle with discrete tasks (e.g. mathematics) and interacting with the outside world (e.g. getting the weather). This is where tools come in.

Tools are actions that an LLM can invoke. The results of these actions can be reported back to the LLM to be considered in the next response.

For example, if a user asks about the current weather, without tools, the model would only be able to provide general information based on its training data. But with a weather tool, it can fetch and provide up-to-date, location-specific weather information.

Let's enhance your chatbot by adding a simple weather tool.

Update Your API Route

Modify your `src/routes/api/chat/+server.ts` file to include the new weather tool:

```
import { createOpenAI } from '@ai-sdk/openai';

import {
  streamText,
  type UIMessage,
```

```
  convertToModelMessages,
  tool,
  stepCountIs,
} from 'ai';

import { z } from 'zod';

import { OPENAI_API_KEY } from '$env/static/private';

const openai = createOpenAI({
  apiKey: OPENAI_API_KEY,
});

export async function POST({ request }) {
  const { messages }: { messages: UIMessage[] } = await request.json();

  const result = streamText({
    model: openai('gpt-4o'),
    messages: convertToModelMessages(messages),
    tools: {
      weather: tool({
        description: 'Get the weather in a location (fahrenheit)',
        inputSchema: z.object({
          location: z.string().describe('The location to get the weather for'),
        }),
        execute: async ({ location }) => {
          const temperature = Math.round(Math.random() * (90 - 32) + 32);
          return {
            location,
            temperature,
          };
        },
      }),
    },
  });

  return result.toUIMessageStreamResponse();
}
```

In this updated code:

1. You import the `tool` function from the `ai` package and `z` from `zod` for schema validation.
2. You define a `tools` object with a `weather` tool. This tool:
    - Has a description that helps the model understand when to use it.
    - Defines `inputSchema` using a Zod schema, specifying that it requires a `location` string to execute this tool. The model will attempt to extract this input from the context of the conversation. If it can't, it will ask the user for the missing information.
    - Defines an `execute` function that simulates getting weather data (in this case, it returns a random temperature). This is an asynchronous function running on the server so you can fetch real data from an external API.

Now your chatbot can "fetch" weather information for any location the user asks about. When the model determines it needs to use the weather tool, it will generate a tool call with the necessary input. The `execute` function will then be automatically run, and the tool output will be added to the `messages` as a `tool` message.

Try asking something like "What's the weather in New York?" and see how the model uses the new tool.

Notice the blank response in the UI? This is because instead of generating a text response, the model generated a tool call. You can access the tool call and subsequent tool result on the client via the `tool-weather` part of the `message.parts` array.

> Tool parts are always named `tool-{toolName}`, where `{toolName}` is the key you used when defining the tool. In this case, since we defined the tool as `weather`, the part type is `tool-weather`.

Update the UI

To display the tool invocation in your UI, update your `src/routes/+page.svelte` file:

```
<script lang="ts">
  import { Chat } from '@ai-sdk/svelte';

  let input = '';
  const chat = new Chat({});

  function handleSubmit(event: SubmitEvent) {
    event.preventDefault();
    chat.sendMessage({ text: input });
    input = '';
  }
</script>

<main>
  <ul>
    {#each chat.messages as message, messageIndex (messageIndex)}
      <li>
        <div>{message.role}</div>
        <div>
          {#each message.parts as part, partIndex (partIndex)}
            {#if part.type === 'text'}
              <div>{part.text}</div>
            {:else if part.type === 'tool-weather'}
              <pre>{JSON.stringify(part, null, 2)}</pre>
            {/if}
          {/each}
        </div>
      </li>
    {/each}
  </ul>
  <form onsubmit={handleSubmit}>
    <input bind:value={input} />
    <button type="submit">Send</button>
  </form>
</main>
```

With this change, you're updating the UI to handle different message parts. For text parts, you display the text content as before. For weather tool invocations, you display a JSON representation of the tool call and its result.

Now, when you ask about the weather, you'll see the tool call and its result displayed in your chat interface.

Enabling Multi-Step Tool Calls

You may have noticed that while the tool is now visible in the chat interface, the model isn't using this information to answer your original query. This is because once the model generates a tool call, it has technically completed its generation.

To solve this, you can enable multi-step tool calls using `stopWhen`. By default, `stopWhen` is set to `stepCountIs(1)`, which means generation stops after the first step when there are tool results. By changing this condition, you can allow the model to automatically send tool results back to itself to trigger additional generations until your specified stopping condition is met. In this case, you want the model to continue generating so it can use the weather tool results to answer your original question.

Update Your API Route

Modify your `src/routes/api/chat/+server.ts` file to include the `stopWhen` condition:

```
import { createOpenAI } from '@ai-sdk/openai';

import {
  streamText,
  type UIMessage,
  convertToModelMessages,
  tool,
  stepCountIs,
} from 'ai';

import { z } from 'zod';

import { OPENAI_API_KEY } from '$env/static/private';

const openai = createOpenAI({
  apiKey: OPENAI_API_KEY,
});

export async function POST({ request }) {
  const { messages }: { messages: UIMessage[] } = await request.json();

  const result = streamText({
    model: openai('gpt-4o'),
    messages: convertToModelMessages(messages),
    stopWhen: stepCountIs(5),
    tools: {
      weather: tool({
        description: 'Get the weather in a location (fahrenheit)',
        inputSchema: z.object({
          location: z.string().describe('The location to get the weather for'),
        }),
        execute: async ({ location }) => {
          const temperature = Math.round(Math.random() * (90 - 32) + 32);
          return {
            location,
```

```
            temperature,
          };
        },
      }),
    },
  });

  return result.toUIMessageStreamResponse();
}
```

Head back to the browser and ask about the weather in a location. You should now see the model using the weather tool results to answer your question.

By setting `stopWhen: stepCountIs(5)`, you're allowing the model to use up to 5 "steps" for any given generation. This enables more complex interactions and allows the model to gather and process information over several steps if needed. You can see this in action by adding another tool to convert the temperature from Fahrenheit to Celsius.

Add another tool

Update your `src/routes/api/chat/+server.ts` file to add a new tool to convert the temperature from Fahrenheit to Celsius:

```
import { createOpenAI } from '@ai-sdk/openai';

import {
  streamText,
  type UIMessage,
  convertToModelMessages,
  tool,
  stepCountIs,
} from 'ai';

import { z } from 'zod';

import { OPENAI_API_KEY } from '$env/static/private';

const openai = createOpenAI({
  apiKey: OPENAI_API_KEY,
});

export async function POST({ request }) {
  const { messages }: { messages: UIMessage[] } = await request.json();

  const result = streamText({
    model: openai('gpt-4o'),
    messages: convertToModelMessages(messages),
    stopWhen: stepCountIs(5),
    tools: {
      weather: tool({
        description: 'Get the weather in a location (fahrenheit)',
        inputSchema: z.object({
          location: z.string().describe('The location to get the weather for'),
        }),
        execute: async ({ location }) => {
```

```
          const temperature = Math.round(Math.random() * (90 - 32) + 32);
          return {
            location,
            temperature,
          };
        },
      }),
      convertFahrenheitToCelsius: tool({
        description: 'Convert a temperature in fahrenheit to celsius',
        inputSchema: z.object({
          temperature: z
            .number()
            .describe('The temperature in fahrenheit to convert'),
        }),
        execute: async ({ temperature }) => {
          const celsius = Math.round((temperature - 32) * (5 / 9));
          return {
            celsius,
          };
        },
      }),
    },
  });

  return result.toUIMessageStreamResponse();
}
```

Update Your Frontend

Update your UI to handle the new temperature conversion tool by modifying the tool part handling:

```
<script lang="ts">
  import { Chat } from '@ai-sdk/svelte';

  let input = '';
  const chat = new Chat({});

  function handleSubmit(event: SubmitEvent) {
    event.preventDefault();
    chat.sendMessage({ text: input });
    input = '';
  }
</script>

<main>
  <ul>
    {#each chat.messages as message, messageIndex (messageIndex)}
      <li>
        <div>{message.role}</div>
        <div>
          {#each message.parts as part, partIndex (partIndex)}
            {#if part.type === 'text'}
              <div>{part.text}</div>
            {:else if part.type === 'tool-weather' || part.type === 'tool-convert
```

```
              <pre>{JSON.stringify(part, null, 2)}</pre>
            {/if}
          {/each}
        </div>
      </li>
    {/each}
  </ul>
  <form onsubmit={handleSubmit}>
    <input bind:value={input} />
    <button type="submit">Send</button>
  </form>
</main>
```

This update handles the new `tool-convertFahrenheitToCelsius` part type, displaying the temperature conversion tool calls and results in the UI.

Now, when you ask "What's the weather in New York in celsius?", you should see a more complete interaction:

1. The model will call the weather tool for New York.
2. You'll see the tool output displayed.
3. It will then call the temperature conversion tool to convert the temperature from Fahrenheit to Celsius.
4. The model will then use that information to provide a natural language response about the weather in New York.

This multi-step approach allows the model to gather information and use it to provide more accurate and contextual responses, making your chatbot considerably more useful.

This simple example demonstrates how tools can expand your model's capabilities. You can create more complex tools to integrate with real APIs, databases, or any other external systems, allowing the model to access and process real-world data in real-time. Tools bridge the gap between the model's knowledge cutoff and current information.

How does `@ai-sdk/svelte` differ from `@ai-sdk/react`?

The surface-level difference is that Svelte uses classes to manage state, whereas React uses hooks, so `useChat` in React is `Chat` in Svelte. Other than that, there are a few things to keep in mind:

1. Arguments to classes aren't reactive by default

Unlike in React, where hooks are rerun any time their containing component is invalidated, code in the `script` block of a Svelte component is only run once when the component is created.
This means that, if you want arguments to your class to be reactive, you need to make sure you pass a *reference* into the class, rather than a value:

```
<script>
  import { Chat } from '@ai-sdk/svelte';

  let { id } = $props;

  // won't work; the class instance will be created once, `id` will be copied by
  let chat = new Chat({ id });
```

```
  // will work; passes `id` by reference, so `Chat` always has the latest value
  let chat = new Chat({
    get id() {
      return id;
    },
  });
</script>
```

Keep in mind that this normally doesn't matter; most parameters you'll pass into the Chat class are static (for example, you typically wouldn't expect your `onError` handler to change).

2. You can't destructure class properties

In vanilla JavaScript, destructuring class properties copies them by value and "disconnects" them from their class instance:

```
const classInstance = new Whatever();
classInstance.foo = 'bar';

const { foo } = classInstance;
classInstance.foo = 'baz';

console.log(foo); // 'bar'
```

The same is true of classes in Svelte:

```
<script>
  import { Chat } from '@ai-sdk/svelte';

  const chat = new Chat({});
  let { messages } = chat;

  chat.append({ content: 'Hello, world!', role: 'user' }).then(() => {
    console.log(messages); // []
    console.log(chat.messages); // [{ content: 'Hello, world!', role: 'user' }] (
  });
</script>
```

3. Instance synchronization requires context

In React, hook instances with the same `id` are synchronized -- so two instances of `useChat` will have the same `messages`, `status`, etc. if they have the same `id`.
For most use cases, you probably don't need this behavior -- but if you do, you can create a context in your root layout file using `createAIContext`:

```
<script>
  import { createAIContext } from '@ai-sdk/svelte';

  let { children } = $props;

  createAIContext();

  // all hooks created after this or in components that are children of this comp
  // will have synchronized state
```

```
</script>

{@render children()}
```

Where to Next?

You've built an AI chatbot using the AI SDK! From here, you have several paths to explore:

- To learn more about the AI SDK, read through the documentation.
- If you're interested in diving deeper with guides, check out the RAG (retrieval-augmented generation) and multi-modal chatbot guides.
- To jumpstart your first AI project, explore available templates.
- To learn more about Svelte, check out the official documentation.

Vue.js (Nuxt) Quickstart

The AI SDK is a powerful Typescript library designed to help developers build AI-powered applications.

In this quickstart tutorial, you'll build a simple AI-chatbot with a streaming user interface. Along the way, you'll learn key concepts and techniques that are fundamental to using the SDK in your own projects.

If you are unfamiliar with the concepts of Prompt Engineering and HTTP Streaming, you can optionally read these documents first.

Prerequisites

To follow this quickstart, you'll need:

- Node.js 18+ and pnpm installed on your local development machine.
- An OpenAI API key.

If you haven't obtained your OpenAI API key, you can do so by signing up on the OpenAI website.

Setup Your Application

Start by creating a new Nuxt application. This command will create a new directory named `my-ai-app` and set up a basic Nuxt application inside it.

```
pnpm create nuxt my-ai-app
```

Navigate to the newly created directory:

```
cd my-ai-app
```

Install dependencies

Install `ai` and `@ai-sdk/openai`, the AI SDK's OpenAI provider.

> The AI SDK is designed to be a unified interface to interact with any large language model. This means that you can change model and providers with just one line of code! Learn more about available providers and building custom providers in the providers section.

Using pnpm:

```
pnpm add ai @ai-sdk/openai @ai-sdk/vue zod
```

Configure OpenAI API key

Create a `.env` file in your project root and add your OpenAI API Key. This key is used to authenticate your application with the OpenAI service.

```
touch .env
```

Edit the `.env` file:

```
NUXT_OPENAI_API_KEY=xxxxxxxxx
```

Replace `xxxxxxxxx` with your actual OpenAI API key and configure the environment variable in `nuxt.config.ts`:

```
export default defineNuxtConfig({
  // rest of your nuxt config

  runtimeConfig: {
    openaiApiKey: '',
  },
});
```

> The AI SDK's OpenAI Provider will default to using the `OPENAI_API_KEY` environment variable.

Create an API route

Create an API route, `server/api/chat.ts` and add the following code:

```
import { streamText, UIMessage, convertToModelMessages } from 'ai';
import { createOpenAI } from '@ai-sdk/openai';

export default defineLazyEventHandler(async () => {
  const apiKey = useRuntimeConfig().openaiApiKey;

  if (!apiKey) throw new Error('Missing OpenAI API key');

  const openai = createOpenAI({
    apiKey: apiKey,
  });

  return defineEventHandler(async (event: any) => {
    const { messages }: { messages: UIMessage[] } = await readBody(event);

    const result = streamText({
      model: openai('gpt-4o'),
      messages: convertToModelMessages(messages),
    });

    return result.toUIMessageStreamResponse();
  });
});
```

Let's take a look at what is happening in this code:

1. Create an OpenAI provider instance with the `createOpenAI` function from the `@ai-sdk/openai` package.
2. Define an Event Handler and extract `messages` from the body of the request. The `messages` variable contains a history of the conversation between you and the chatbot and provides the chatbot with the necessary context to make the next generation. The `messages` are of UIMessage type, which are designed for use in application UI - they contain the entire message history and associated metadata like timestamps.
3. Call `streamText`, which is imported from the `ai` package. This function accepts a configuration object that contains a `model` provider (defined in step 1) and `messages`

(defined in step 2). You can pass additional settings to further customise the model's behaviour. The `messages` key expects a `ModelMessage[]` array. This type is different from `UIMessage` in that it does not include metadata, such as timestamps or sender information. To convert between these types, we use the `convertToModelMessages` function, which strips the UI-specific metadata and transforms the `UIMessage[]` array into the `ModelMessage[]` format that the model expects.

4. The `streamText` function returns a `StreamTextResult`. This result object contains the `toDataStreamResponse` function which converts the result to a streamed response object.
5. Return the result to the client to stream the response.

Wire up the UI

Now that you have an API route that can query an LLM, it's time to setup your frontend. The AI SDK's UI package abstract the complexity of a chat interface into one hook, `useChat`.

Update your root page (`pages/index.vue`) with the following code to show a list of chat messages and provide a user message input:

```
<script setup lang="ts">
import { Chat } from "@ai-sdk/vue";
import { ref } from "vue";

const input = ref("");
const chat = new Chat({});

const handleSubmit = (e: Event) => {
    e.preventDefault();
    chat.sendMessage({ text: input.value });
    input.value = "";
};
</script>

<template>
    <div>
        <div v-for="(m, index) in chat.messages" :key="m.id ? m.id : index">
            {{ m.role === "user" ? "User: " : "AI: " }}
            <div>
                <div v-for="(part, index) in m.parts" :key="`${m.id}-${part.type}
                    <div v-if="part.type === 'text'">{{ part.text }}</div>
                </div>
            </div>
        </div>

        <form @submit="handleSubmit">
            <input v-model="input" placeholder="Say something..." />
        </form>
    </div>
</template>
```

If your project has `app.vue` instead of `pages/index.vue`, delete the `app.vue` file and create a new `pages/index.vue` file with the code above.

This page utilizes the `useChat` hook, which will, by default, use the API route you created earlier (`/api/chat`). The hook provides functions and state for handling user input and form submission.

The `useChat` hook provides multiple utility functions and state variables:

- `messages` - the current chat messages (an array of objects with `id`, `role`, and `parts` properties).
- `sendMessage` - a function to send a message to the chat API.

The component uses local state (`ref`) to manage the input field value, and handles form submission by calling `sendMessage` with the input text and then clearing the input field.

The LLM's response is accessed through the message `parts` array. Each message contains an ordered array of `parts` that represents everything the model generated in its response. These parts can include plain text, reasoning tokens, and more that you will see later. The `parts` array preserves the sequence of the model's outputs, allowing you to display or process each component in the order it was generated.

Running Your Application

With that, you have built everything you need for your chatbot! To start your application, use the command:

```
pnpm run dev
```

Head to your browser and open http://localhost:3000. You should see an input field. Test it out by entering a message and see the AI chatbot respond in real-time! The AI SDK makes it fast and easy to build AI chat interfaces with Nuxt.

Enhance Your Chatbot with Tools

While large language models (LLMs) have incredible generation capabilities, they struggle with discrete tasks (e.g. mathematics) and interacting with the outside world (e.g. getting the weather). This is where tools come in.

Tools are actions that an LLM can invoke. The results of these actions can be reported back to the LLM to be considered in the next response.

For example, if a user asks about the current weather, without tools, the model would only be able to provide general information based on its training data. But with a weather tool, it can fetch and provide up-to-date, location-specific weather information.

Let's enhance your chatbot by adding a simple weather tool.

Update Your API Route

Modify your `server/api/chat.ts` file to include the new weather tool:

```
import { streamText, UIMessage, convertToModelMessages, tool } from 'ai';
import { createOpenAI } from '@ai-sdk/openai';
import { z } from 'zod';

export default defineLazyEventHandler(async () => {
  const apiKey = useRuntimeConfig().openaiApiKey;

  if (!apiKey) throw new Error('Missing OpenAI API key');

  const openai = createOpenAI({
```

```
    apiKey: apiKey,
  });

  return defineEventHandler(async (event: any) => {
    const { messages }: { messages: UIMessage[] } = await readBody(event);

    const result = streamText({
      model: openai('gpt-4o'),
      messages: convertToModelMessages(messages),
      tools: {
        weather: tool({
          description: 'Get the weather in a location (fahrenheit)',
          inputSchema: z.object({
            location: z.string().describe('The location to get the weather for'),
          }),
          execute: async ({ location }) => {
            const temperature = Math.round(Math.random() * (90 - 32) + 32);

            return {
              location,
              temperature,
            };
          },
        }),
      },
    });

    return result.toUIMessageStreamResponse();
  });
});
```

In this updated code:

1. You import the `tool` function from the `ai` package and `z` from `zod` for schema validation.
2. You define a `tools` object with a `weather` tool. This tool:
   - Has a description that helps the model understand when to use it.
   - Defines `inputSchema` using a Zod schema, specifying that it requires a `location` string to execute this tool. The model will attempt to extract this input from the context of the conversation. If it can't, it will ask the user for the missing information.
   - Defines an `execute` function that simulates getting weather data (in this case, it returns a random temperature). This is an asynchronous function running on the server so you can fetch real data from an external API.

Now your chatbot can "fetch" weather information for any location the user asks about. When the model determines it needs to use the weather tool, it will generate a tool call with the necessary input. The `execute` function will then be automatically run, and the tool output will be added to the `messages` as a `tool` message.

Try asking something like "What's the weather in New York?" and see how the model uses the new tool.

Notice the blank response in the UI? This is because instead of generating a text response, the model generated a tool call. You can access the tool call and subsequent tool result on the client via the `tool-weather` part of the `message.parts` array.

Tool parts are always named `tool-{toolName}`, where `{toolName}` is the key you used when defining the tool. In this case, since we defined the tool as `weather`, the part type is `tool-weather`.

Update the UI

To display the tool invocation in your UI, update your `pages/index.vue` file:

```
<script setup lang="ts">
import { Chat } from "@ai-sdk/vue";
import { ref } from "vue";

const input = ref("");
const chat = new Chat({});

const handleSubmit = (e: Event) => {
    e.preventDefault();
    chat.sendMessage({ text: input.value });
    input.value = "";
};
</script>

<template>
    <div>
        <div v-for="(m, index) in chat.messages" :key="m.id ? m.id : index">
            {{ m.role === "user" ? "User: " : "AI: " }}
            <div>
                <div v-for="(part, index) in m.parts" :key="`${m.id}-${part.type}
                    <div v-if="part.type === 'text'">{{ part.text }}</div>
                    <pre v-if="part.type === 'tool-weather'">{{ JSON.stringify(pa
                </div>
            </div>
        </div>

        <form @submit="handleSubmit">
            <input v-model="input" placeholder="Say something..." />
        </form>
    </div>
</template>
```

With this change, you're updating the UI to handle different message parts. For text parts, you display the text content as before. For weather tool invocations, you display a JSON representation of the tool call and its result.

Now, when you ask about the weather, you'll see the tool call and its result displayed in your chat interface.

Enabling Multi-Step Tool Calls

You may have noticed that while the tool is now visible in the chat interface, the model isn't using this information to answer your original query. This is because once the model generates a tool call, it has technically completed its generation.

To solve this, you can enable multi-step tool calls using `stopWhen`. By default, `stopWhen` is set to `stepCountIs(1)`, which means generation stops after the first step when there are tool results. By

changing this condition, you can allow the model to automatically send tool results back to itself to trigger additional generations until your specified stopping condition is met. In this case, you want the model to continue generating so it can use the weather tool results to answer your original question.

Update Your API Route

Modify your `server/api/chat.ts` file to include the `stopWhen` condition:

```
import {
  streamText,
  UIMessage,
  convertToModelMessages,
  tool,
  stepCountIs,
} from 'ai';
import { createOpenAI } from '@ai-sdk/openai';
import { z } from 'zod';

export default defineLazyEventHandler(async () => {
  const apiKey = useRuntimeConfig().openaiApiKey;

  if (!apiKey) throw new Error('Missing OpenAI API key');

  const openai = createOpenAI({
    apiKey: apiKey,
  });

  return defineEventHandler(async (event: any) => {
    const { messages }: { messages: UIMessage[] } = await readBody(event);

    const result = streamText({
      model: openai('gpt-4o'),
      messages: convertToModelMessages(messages),
      stopWhen: stepCountIs(5),
      tools: {
        weather: tool({
          description: 'Get the weather in a location (fahrenheit)',
          inputSchema: z.object({
            location: z.string().describe('The location to get the weather for'),
          }),
          execute: async ({ location }) => {
            const temperature = Math.round(Math.random() * (90 - 32) + 32);

            return {
              location,
              temperature,
            };
          },
        }),
      },
    });

    return result.toUIMessageStreamResponse();
```

```
  });
});
```

Head back to the browser and ask about the weather in a location. You should now see the model using the weather tool results to answer your question.

By setting `stopWhen: stepCountIs(5)`, you're allowing the model to use up to 5 "steps" for any given generation. This enables more complex interactions and allows the model to gather and process information over several steps if needed. You can see this in action by adding another tool to convert the temperature from Fahrenheit to Celsius.

Add another tool

Update your `server/api/chat.ts` file to add a new tool to convert the temperature from Fahrenheit to Celsius:

```
import {
  streamText,
  UIMessage,
  convertToModelMessages,
  tool,
  stepCountIs,
} from 'ai';
import { createOpenAI } from '@ai-sdk/openai';
import { z } from 'zod';

export default defineLazyEventHandler(async () => {
  const apiKey = useRuntimeConfig().openaiApiKey;

  if (!apiKey) throw new Error('Missing OpenAI API key');

  const openai = createOpenAI({
    apiKey: apiKey,
  });

  return defineEventHandler(async (event: any) => {
    const { messages }: { messages: UIMessage[] } = await readBody(event);

    const result = streamText({
      model: openai('gpt-4o'),
      messages: convertToModelMessages(messages),
      stopWhen: stepCountIs(5),
      tools: {
        weather: tool({
          description: 'Get the weather in a location (fahrenheit)',
          inputSchema: z.object({
            location: z.string().describe('The location to get the weather for'),
          }),
          execute: async ({ location }) => {
            const temperature = Math.round(Math.random() * (90 - 32) + 32);

            return {
              location,
              temperature,
            };
```

```
        },
      }),
      convertFahrenheitToCelsius: tool({
        description: 'Convert a temperature in fahrenheit to celsius',
        inputSchema: z.object({
          temperature: z.number().describe('The temperature in fahrenheit to co
        }),
        execute: async ({ temperature }) => {
          const celsius = Math.round((temperature - 32) * (5 / 9));

          return {
            celsius,
          };
        },
      }),
    },
  });

  return result.toUIMessageStreamResponse();
 });
});
```

Update Your Frontend

Update your UI to handle the new temperature conversion tool by modifying the tool part handling:

```
<script setup lang="ts">
import { Chat } from "@ai-sdk/vue";
import { ref } from "vue";

const input = ref("");
const chat = new Chat({});

const handleSubmit = (e: Event) => {
    e.preventDefault();
    chat.sendMessage({ text: input.value });
    input.value = "";
};
</script>

<template>
    <div>
        <div v-for="(m, index) in chat.messages" :key="m.id ? m.id : index">
            {{ m.role === "user" ? "User: " : "AI: " }}
            <div>
                <div v-for="(part, index) in m.parts" :key="`${m.id}-${part.type}
                    <div v-if="part.type === 'text'">{{ part.text }}</div>
                    <pre
                        v-if="part.type === 'tool-weather' || part.type === 'tool
                    >{{ JSON.stringify(part, null, 2) }}</pre>
                </div>
            </div>
        </div>
```

```
        <form @submit="handleSubmit">
            <input v-model="input" placeholder="Say something..." />
        </form>
    </div>
</template>
```

This update handles the new `tool-convertFahrenheitToCelsius` part type, displaying the temperature conversion tool calls and results in the UI.

Now, when you ask "What's the weather in New York in celsius?", you should see a more complete interaction:

1. The model will call the weather tool for New York.
2. You'll see the tool output displayed.
3. It will then call the temperature conversion tool to convert the temperature from Fahrenheit to Celsius.
4. The model will then use that information to provide a natural language response about the weather in New York.

This multi-step approach allows the model to gather information and use it to provide more accurate and contextual responses, making your chatbot considerably more useful.

This simple example demonstrates how tools can expand your model's capabilities. You can create more complex tools to integrate with real APIs, databases, or any other external systems, allowing the model to access and process real-world data in real-time. Tools bridge the gap between the model's knowledge cutoff and current information.

Where to Next?

You've built an AI chatbot using the AI SDK! From here, you have several paths to explore:

- To learn more about the AI SDK, read through the documentation.
- If you're interested in diving deeper with guides, check out the RAG (retrieval-augmented generation) and multi-modal chatbot guides.
- To jumpstart your first AI project, explore available templates.

Previous: Svelte | Next: Node.js

Node.js Quickstart

The AI SDK is a powerful Typescript library designed to help developers build AI-powered applications.

In this quickstart tutorial, you'll build a simple AI-chatbot with a streaming user interface. Along the way, you'll learn key concepts and techniques that are fundamental to using the SDK in your own projects.

If you are unfamiliar with the concepts of Prompt Engineering and HTTP Streaming, you can optionally read these documents first.

Prerequisites

To follow this quickstart, you'll need:

- Node.js 18+ and pnpm installed on your local development machine.
- An OpenAI API key.

If you haven't obtained your OpenAI API key, you can do so by signing up on the OpenAI website.

Setup Your Application

Start by creating a new directory using the `mkdir` command. Change into your new directory and then run the `pnpm init` command. This will create a `package.json` in your new directory.

```
mkdir my-ai-app
cd my-ai-app
pnpm init
```

Install Dependencies

Install `ai` and `@ai-sdk/openai`, the AI SDK's OpenAI provider, along with other necessary dependencies.

> The AI SDK is designed to be a unified interface to interact with any large language model. This means that you can change model and providers with just one line of code! Learn more about available providers and building custom providers in the providers section.

```
pnpm add ai@beta @ai-sdk/openai@beta zod dotenv
pnpm add -D @types/node tsx typescript
```

The `ai` and `@ai-sdk/openai` packages contain the AI SDK and the AI SDK OpenAI provider, respectively. You will use `zod` to define type-safe schemas that you will pass to the large language model (LLM). You will use `dotenv` to access environment variables (your OpenAI key) within your application. There are also three development dependencies, installed with the `-D` flag, that are necessary to run your Typescript code.

Configure OpenAI API key

Create a `.env` file in your project's root directory and add your OpenAI API Key. This key is used to authenticate your application with the OpenAI service.

```
touch .env
```

Edit the `.env` file:

```
OPENAI_API_KEY=xxxxxxxxx
```

Replace `xxxxxxxxx` with your actual OpenAI API key.

> The AI SDK's OpenAI Provider will default to using the `OPENAI_API_KEY` environment variable.

Create Your Application

Create an `index.ts` file in the root of your project and add the following code:

```
import { openai } from '@ai-sdk/openai';

import { ModelMessage, streamText } from 'ai';

import 'dotenv/config';

import * as readline from 'node:readline/promises';

const terminal = readline.createInterface({
  input: process.stdin,
  output: process.stdout,
});

const messages: ModelMessage[] = [];

async function main() {
  while (true) {
    const userInput = await terminal.question('You: ');

    messages.push({ role: 'user', content: userInput });

    const result = streamText({
      model: openai('gpt-4o'),
      messages,
    });

    let fullResponse = '';
    process.stdout.write('\nAssistant: ');
    for await (const delta of result.textStream) {
      fullResponse += delta;
      process.stdout.write(delta);
    }
    process.stdout.write('\n\n');

    messages.push({ role: 'assistant', content: fullResponse });
  }
}

main().catch(console.error);
```

Let's take a look at what is happening in this code:

1. Set up a readline interface to take input from the terminal, enabling interactive sessions directly from the command line.
2. Initialize an array called `messages` to store the history of your conversation. This history allows the model to maintain context in ongoing dialogues.
3. In the `main` function:

- Prompt for and capture user input, storing it in `userInput`.
- Add user input to the `messages` array as a user message.
- Call `streamText`, which is imported from the `ai` package. This function accepts a configuration object that contains a `model` provider and `messages`.
- Iterate over the text stream returned by the `streamText` function (`result.textStream`) and print the contents of the stream to the terminal.
- Add the assistant's response to the `messages` array.

Running Your Application

With that, you have built everything you need for your chatbot! To start your application, use the command:

```
pnpm tsx index.ts
```

You should see a prompt in your terminal. Test it out by entering a message and see the AI chatbot respond in real-time! The AI SDK makes it fast and easy to build AI chat interfaces with Node.js.

Enhance Your Chatbot with Tools

While large language models (LLMs) have incredible generation capabilities, they struggle with discrete tasks (e.g. mathematics) and interacting with the outside world (e.g. getting the weather). This is where tools come in.

Tools are actions that an LLM can invoke. The results of these actions can be reported back to the LLM to be considered in the next response.

For example, if a user asks about the current weather, without tools, the model would only be able to provide general information based on its training data. But with a weather tool, it can fetch and provide up-to-date, location-specific weather information.

Let's enhance your chatbot by adding a simple weather tool.

Update Your Application

Modify your `index.ts` file to include the new weather tool:

```
import { openai } from '@ai-sdk/openai';

import { ModelMessage, streamText, tool } from 'ai';

import 'dotenv/config';

import { z } from 'zod';

import * as readline from 'node:readline/promises';
```

```
const terminal = readline.createInterface({
  input: process.stdin,
  output: process.stdout,
});

const messages: ModelMessage[] = [];

async function main() {
  while (true) {
    const userInput = await terminal.question('You: ');

    messages.push({ role: 'user', content: userInput });

    const result = streamText({
      model: openai('gpt-4o'),
      messages,
      tools: {
        weather: tool({
          description: 'Get the weather in a location (fahrenheit)',
          inputSchema: z.object({
            location: z
              .string()
              .describe('The location to get the weather for'),
          }),
          execute: async ({ location }) => {
            const temperature = Math.round(
              Math.random() * (90 - 32) + 32
            );
            return {
              location,
              temperature,
            };
          },
        }),
      },
    });

    let fullResponse = '';
    process.stdout.write('\nAssistant: ');
    for await (const delta of result.textStream) {
      fullResponse += delta;
      process.stdout.write(delta);
    }
    process.stdout.write('\n\n');

    messages.push({ role: 'assistant', content: fullResponse });
  }
}

main().catch(console.error);
```

In this updated code:

1. You import the `tool` function from the `ai` package.
2. You define a `tools` object with a `weather` tool. This tool:

- Has a description that helps the model understand when to use it.
- Defines `inputSchema` using a Zod schema, specifying that it requires a `location` string to execute this tool. The model will attempt to extract this input from the context of the conversation. If it can't, it will ask the user for the missing information.
- Defines an `execute` function that simulates getting weather data (in this case, it returns a random temperature). This is an asynchronous function running on the server so you can fetch real data from an external API.

Now your chatbot can "fetch" weather information for any location the user asks about. When the model determines it needs to use the weather tool, it will generate a tool call with the necessary parameters. The `execute` function will then be automatically run, and the results will be used by the model to generate its response.

Try asking something like "What's the weather in New York?" and see how the model uses the new tool.

Notice the blank "assistant" response? This is because instead of generating a text response, the model generated a tool call. You can access the tool call and subsequent tool result in the `toolCall` and `toolResult` keys of the result object.

```
import { openai } from '@ai-sdk/openai';

import { ModelMessage, streamText, tool } from 'ai';

import 'dotenv/config';

import { z } from 'zod';

import * as readline from 'node:readline/promises';

const terminal = readline.createInterface({
  input: process.stdin,
  output: process.stdout,
});

const messages: ModelMessage[] = [];

async function main() {
  while (true) {
    const userInput = await terminal.question('You: ');

    messages.push({ role: 'user', content: userInput });

    const result = streamText({
      model: openai('gpt-4o'),
      messages,
      tools: {
        weather: tool({
          description: 'Get the weather in a location (fahrenheit)',
          inputSchema: z.object({
            location: z
              .string()
              .describe('The location to get the weather for'),
          }),
          execute: async ({ location }) => {
            const temperature = Math.round(
```

```
              Math.random() * (90 - 32) + 32
            );
            return {
              location,
              temperature,
            };
          },
        }),
      },
    });

    let fullResponse = '';
    process.stdout.write('\nAssistant: ');
    for await (const delta of result.textStream) {
      fullResponse += delta;
      process.stdout.write(delta);
    }
    process.stdout.write('\n\n');

    console.log(await result.toolCalls);
    console.log(await result.toolResults);

    messages.push({ role: 'assistant', content: fullResponse });
  }
}

main().catch(console.error);
```

Now, when you ask about the weather, you'll see the tool call and its result displayed in your chat interface.

Enabling Multi-Step Tool Calls

You may have noticed that while the tool results are visible in the chat interface, the model isn't using this information to answer your original query. This is because once the model generates a tool call, it has technically completed its generation.

To solve this, you can enable multi-step tool calls using `stopWhen`. This feature will automatically send tool results back to the model to trigger an additional generation until the stopping condition you define is met. In this case, you want the model to answer your question using the results from the weather tool.

Update Your Application

Modify your `index.ts` file to configure stopping conditions with `stopWhen`:

```
import { openai } from '@ai-sdk/openai';

import { ModelMessage, streamText, tool, stepCountIs } from 'ai';

import 'dotenv/config';

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

```
import * as readline from 'node:readline/promises';

const terminal = readline.createInterface({
  input: process.stdin,
  output: process.stdout,
});

const messages: ModelMessage[] = [];

async function main() {
  while (true) {
    const userInput = await terminal.question('You: ');

    messages.push({ role: 'user', content: userInput });

    const result = streamText({
      model: openai('gpt-4o'),
      messages,
      tools: {
        weather: tool({
          description: 'Get the weather in a location (fahrenheit)',
          inputSchema: z.object({
            location: z
              .string()
              .describe('The location to get the weather for'),
          }),
          execute: async ({ location }) => {
            const temperature = Math.round(
              Math.random() * (90 - 32) + 32
            );
            return {
              location,
              temperature,
            };
          },
        }),
      },
      stopWhen: stepCountIs(5),
      onStepFinish: async ({ toolResults }) => {
        if (toolResults.length) {
          console.log(JSON.stringify(toolResults, null, 2));
        }
      },
    });

    let fullResponse = '';
    process.stdout.write('\nAssistant: ');
    for await (const delta of result.textStream) {
      fullResponse += delta;
      process.stdout.write(delta);
    }
    process.stdout.write('\n\n');

    messages.push({ role: 'assistant', content: fullResponse });
```

```
  }
}

main().catch(console.error);
```

In this updated code:

1. You set `stopWhen` to be when `stepCountIs` 5, allowing the model to use up to 5 "steps" for any given generation.
2. You add an `onStepFinish` callback to log any `toolResults` from each step of the interaction, helping you understand the model's tool usage. This means we can also delete the `toolCall` and `toolResult` `console.log` statements from the previous example.

Now, when you ask about the weather in a location, you should see the model using the weather tool results to answer your question.

By setting `stopWhen: stepCountIs(5)`, you're allowing the model to use up to 5 "steps" for any given generation. This enables more complex interactions and allows the model to gather and process information over several steps if needed. You can see this in action by adding another tool to convert the temperature from Celsius to Fahrenheit.

Adding a second tool

Update your `index.ts` file to add a new tool to convert the temperature from Celsius to Fahrenheit:

```
import { openai } from '@ai-sdk/openai';

import { ModelMessage, streamText, tool, stepCountIs } from 'ai';

import 'dotenv/config';

import { z } from 'zod';

import * as readline from 'node:readline/promises';

const terminal = readline.createInterface({
  input: process.stdin,
  output: process.stdout,
});

const messages: ModelMessage[] = [];

async function main() {
  while (true) {
    const userInput = await terminal.question('You: ');

    messages.push({ role: 'user', content: userInput });

    const result = streamText({
      model: openai('gpt-4o'),
      messages,
      tools: {
        weather: tool({
          description: 'Get the weather in a location (fahrenheit)',
          inputSchema: z.object({
```

```
            location: z
              .string()
              .describe('The location to get the weather for'),
          }),
          execute: async ({ location }) => {
            const temperature = Math.round(
              Math.random() * (90 - 32) + 32
            );
            return {
              location,
              temperature,
            };
          },
        }),
        convertFahrenheitToCelsius: tool({
          description: 'Convert a temperature in fahrenheit to celsius',
          inputSchema: z.object({
            temperature: z
              .number()
              .describe('The temperature in fahrenheit to convert'),
          }),
          execute: async ({ temperature }) => {
            const celsius = Math.round(
              (temperature - 32) * (5 / 9)
            );
            return {
              celsius,
            };
          },
        }),
      },
      stopWhen: stepCountIs(5),
      onStepFinish: async ({ toolResults }) => {
        if (toolResults.length) {
          console.log(JSON.stringify(toolResults, null, 2));
        }
      },
    });

    let fullResponse = '';
    process.stdout.write('\nAssistant: ');
    for await (const delta of result.textStream) {
      fullResponse += delta;
      process.stdout.write(delta);
    }
    process.stdout.write('\n\n');

    messages.push({ role: 'assistant', content: fullResponse });
  }
}

main().catch(console.error);
```

Now, when you ask "What's the weather in New York in celsius?", you should see a more complete interaction:

1. The model will call the weather tool for New York.
2. You'll see the tool result logged.
3. It will then call the temperature conversion tool to convert the temperature from Fahrenheit to Celsius.
4. The model will then use that information to provide a natural language response about the weather in New York.

This multi-step approach allows the model to gather information and use it to provide more accurate and contextual responses, making your chatbot considerably more useful.

This example demonstrates how tools can expand your model's capabilities. You can create more complex tools to integrate with real APIs, databases, or any other external systems, allowing the model to access and process real-world data in real-time and perform actions that interact with the outside world. Tools bridge the gap between the model's knowledge cutoff and current information, while also enabling it to take meaningful actions beyond just generating text responses.

Where to Next?

You've built an AI chatbot using the AI SDK! From here, you have several paths to explore:

- To learn more about the AI SDK, read through the documentation.
- If you're interested in diving deeper with guides, check out the RAG (retrieval-augmented generation) and multi-modal chatbot guides.
- To jumpstart your first AI project, explore available templates.

Previous: Vue.js (Nuxt) | Next: Expo

Expo Quickstart

In this quickstart tutorial, you'll build a simple AI-chatbot with a streaming user interface with Expo. Along the way, you'll learn key concepts and techniques that are fundamental to using the SDK in your own projects.

If you are unfamiliar with the concepts of Prompt Engineering and HTTP Streaming, you can optionally read these documents first.

Prerequisites

To follow this quickstart, you'll need:

- Node.js 18+ and pnpm installed on your local development machine.
- An OpenAI API key.

If you haven't obtained your OpenAI API key, you can do so by signing up on the OpenAI website.

Create Your Application

Start by creating a new Expo application. This command will create a new directory named `my-ai-app` and set up a basic Expo application inside it.

```
pnpm create expo-app@latest my-ai-app
```

Navigate to the newly created directory:

```
cd my-ai-app
```

This guide requires Expo 52 or higher.

Install dependencies

Install `ai`, `@ai-sdk/react` and `@ai-sdk/openai`, the AI package, the AI React package and AI SDK's OpenAI provider respectively.

> The AI SDK is designed to be a unified interface to interact with any large language model. This means that you can change model and providers with just one line of code! Learn more about available providers and building custom providers in the providers section.

```
pnpm add ai @ai-sdk/openai @ai-sdk/react zod
```

Configure OpenAI API key

Create a `.env.local` file in your project root and add your OpenAI API Key. This key is used to authenticate your application with the OpenAI service.

```
touch .env.local
```

Edit the `.env.local` file:

```
OPENAI_API_KEY=xxxxxxxxx
```

Replace `xxxxxxxxx` with your actual OpenAI API key.

The AI SDK's OpenAI Provider will default to using the `OPENAI_API_KEY` environment variable.

Create an API Route

Create a route handler, `app/api/chat+api.ts` and add the following code:

```
import { openai } from '@ai-sdk/openai';

import {
  streamText,
  UIMessage,
  convertToModelMessages,
} from 'ai';

export async function POST(req: Request) {
  const { messages }: { messages: UIMessage[] } = await req.json();

  const result = streamText({
    model: openai('gpt-4o'),
    messages: convertToModelMessages(messages),
  });

  return result.toUIMessageStreamResponse({
    headers: {
      'Content-Type': 'application/octet-stream',
      'Content-Encoding': 'none',
    },
  });
}
```

Let's take a look at what is happening in this code:

1. Define an asynchronous `POST` request handler and extract `messages` from the body of the request. The `messages` variable contains a history of the conversation between you and the chatbot and provides the chatbot with the necessary context to make the next generation.
2. Call `streamText`, which is imported from the `ai` package. This function accepts a configuration object that contains a `model` provider (imported from `@ai-sdk/openai`) and `messages` (defined in step 1). You can pass additional settings to further customise the model's behaviour.
3. The `streamText` function returns a StreamTextResult. This result object contains the toDataStreamResponse function which converts the result to a streamed response object.
4. Finally, return the result to the client to stream the response.

This API route creates a POST request endpoint at `/api/chat`.

Wire up the UI

Now that you have an API route that can query an LLM, it's time to setup your frontend. The AI SDK's UI package abstracts the complexity of a chat interface into one hook, `useChat`.

Update your root page (`app/(tabs)/index.tsx`) with the following code to show a list of chat messages and provide a user message input:

```
import { generateAPIUrl } from '@/utils';

import { useChat } from '@ai-sdk/react';

import { DefaultChatTransport } from 'ai';

import { fetch as expoFetch } from 'expo/fetch';

import { useState } from 'react';

import { View, TextInput, ScrollView, Text, SafeAreaView } from 'react-native';

export default function App() {
  const [input, setInput] = useState('');

  const { messages, error, sendMessage } = useChat({
    transport: new DefaultChatTransport({
      fetch: expoFetch as unknown as typeof globalThis.fetch,
      api: generateAPIUrl('/api/chat'),
    }),
    onError: error => console.error(error, 'ERROR'),
  });

  if (error) return <Text>{error.message}</Text>;

  return (
    <SafeAreaView style={{ height: '100%' }}>
      <View
        style={{
          height: '95%',
          display: 'flex',
          flexDirection: 'column',
          paddingHorizontal: 8,
        }}
      >
        <ScrollView style={{ flex: 1 }}>
          {messages.map(m => (
            <View key={m.id} style={{ marginVertical: 8 }}>
              <View>
                <Text style={{ fontWeight: 700 }}>{m.role}</Text>
                {m.parts.map((part, i) => {
                  switch (part.type) {
                    case 'text':
                      return (
                        <Text key={`${m.id}-${i}`}>
                          {part.text}
                        </Text>
                      );
                  }
                })}
              </View>
```

```
            </View>
          ))}
        </ScrollView>

        <View style={{ marginTop: 8 }}>
          <TextInput
            style={{ backgroundColor: 'white', padding: 8 }}
            placeholder="Say something..."
            value={input}
            onChange={e => setInput(e.nativeEvent.text)}
            onSubmitEditing={e => {
              e.preventDefault();
              sendMessage({ text: input });
              setInput('');
            }}
            autoFocus={true}
          />
        </View>
      </View>
    </SafeAreaView>
  );
}
```

This page utilizes the `useChat` hook, which will, by default, use the `POST` API route you created earlier (`/api/chat`). The hook provides functions and state for handling user input and form submission. The `useChat` hook provides multiple utility functions and state variables:

- `messages` - the current chat messages (an array of objects with `id`, `role`, and `parts` properties).
- `sendMessage` - a function to send a message to the chat API.

The component uses local state (`useState`) to manage the input field value, and handles form submission by calling `sendMessage` with the input text and then clearing the input field.

The LLM's response is accessed through the message `parts` array. Each message contains an ordered array of `parts` that represents everything the model generated in its response. These parts can include plain text, reasoning tokens, and more that you will see later. The `parts` array preserves the sequence of the model's outputs, allowing you to display or process each component in the order it was generated.

> You use the expo/fetch function instead of the native node fetch to enable streaming of chat responses. This requires Expo 52 or higher.

Create the API URL Generator

Because you're using expo/fetch for streaming responses instead of the native fetch function, you'll need an API URL generator to ensure you are using the correct base url and format depending on the client environment (e.g. web or mobile). Create a new file called `utils.ts` in the root of your project and add the following code:

```
import Constants from 'expo-constants';

export const generateAPIUrl = (relativePath: string) => {
  const origin = Constants.experienceUrl.replace('exp://', 'http://');
```

```
  const path = relativePath.startsWith('/')
    ? relativePath
    : `/${relativePath}`;

  if (process.env.NODE_ENV === 'development') {
    return origin.concat(path);
  }

  if (!process.env.EXPO_PUBLIC_API_BASE_URL) {
    throw new Error(
      'EXPO_PUBLIC_API_BASE_URL environment variable is not defined',
    );
  }

  return process.env.EXPO_PUBLIC_API_BASE_URL.concat(path);
};
```

This utility function handles URL generation for both development and production environments, ensuring your API calls work correctly across different devices and configurations.

> Before deploying to production, you must set the `EXPO_PUBLIC_API_BASE_URL` environment variable in your production environment. This variable should point to the base URL of your API server.

Running Your Application

With that, you have built everything you need for your chatbot! To start your application, use the command:

```
pnpm expo
```

Head to your browser and open http://localhost:8081. You should see an input field. Test it out by entering a message and see the AI chatbot respond in real-time! The AI SDK makes it fast and easy to build AI chat interfaces with Expo.

> If you experience "Property `structuredClone` doesn't exist" errors on mobile, add the polyfills described below.

Enhance Your Chatbot with Tools

While large language models (LLMs) have incredible generation capabilities, they struggle with discrete tasks (e.g. mathematics) and interacting with the outside world (e.g. getting the weather). This is where tools come in.

Tools are actions that an LLM can invoke. The results of these actions can be reported back to the LLM to be considered in the next response.

For example, if a user asks about the current weather, without tools, the model would only be able to provide general information based on its training data. But with a weather tool, it can fetch and provide up-to-date, location-specific weather information.

Let's enhance your chatbot by adding a simple weather tool.

Update Your API route

Modify your `app/api/chat+api.ts` file to include the new weather tool:

```
import { openai } from '@ai-sdk/openai';

import {
  streamText,
  UIMessage,
  convertToModelMessages,
  tool,
} from 'ai';

import { z } from 'zod';

export async function POST(req: Request) {
  const { messages }: { messages: UIMessage[] } = await req.json();

  const result = streamText({
    model: openai('gpt-4o'),
    messages: convertToModelMessages(messages),
    tools: {
      weather: tool({
        description: 'Get the weather in a location (fahrenheit)',
        inputSchema: z.object({
          location: z.string().describe('The location to get the weather for'),
        }),
        execute: async ({ location }) => {
          const temperature = Math.round(Math.random() * (90 - 32) + 32);
          return {
            location,
            temperature,
          };
        },
      }),
    },
  });

  return result.toUIMessageStreamResponse({
    headers: {
      'Content-Type': 'application/octet-stream',
      'Content-Encoding': 'none',
    },
  });
}
```

In this updated code:

1. You import the `tool` function from the `ai` package and `z` from `zod` for schema validation.
2. You define a `tools` object with a `weather` tool. This tool:
    - Has a description that helps the model understand when to use it.
    - Defines `inputSchema` using a Zod schema, specifying that it requires a `location` string to execute this tool. The model will attempt to extract this input from the context of the conversation. If it can't, it will ask the user for the missing information.

- Defines an `execute` function that simulates getting weather data (in this case, it returns a random temperature). This is an asynchronous function running on the server so you can fetch real data from an external API.

Now your chatbot can "fetch" weather information for any location the user asks about. When the model determines it needs to use the weather tool, it will generate a tool call with the necessary input. The `execute` function will then be automatically run, and the tool output will be added to the `messages` as a `tool` message.

> You may need to restart your development server for the changes to take effect.

Try asking something like "What's the weather in New York?" and see how the model uses the new tool.

Notice the blank response in the UI? This is because instead of generating a text response, the model generated a tool call. You can access the tool call and subsequent tool result on the client via the `tool-weather` part of the `message.parts` array.

> Tool parts are always named `tool-{toolName}`, where `{toolName}` is the key you used when defining the tool. In this case, since we defined the tool as `weather`, the part type is `tool-weather`.

Update the UI

To display the weather tool invocation in your UI, update your `app/(tabs)/index.tsx` file:

```
import { generateAPIUrl } from '@/utils';

import { useChat } from '@ai-sdk/react';

import { DefaultChatTransport } from 'ai';

import { fetch as expoFetch } from 'expo/fetch';

import { useState } from 'react';

import { View, TextInput, ScrollView, Text, SafeAreaView } from 'react-native';

export default function App() {
  const [input, setInput] = useState('');

  const { messages, error, sendMessage } = useChat({
    transport: new DefaultChatTransport({
      fetch: expoFetch as unknown as typeof globalThis.fetch,
      api: generateAPIUrl('/api/chat'),
    }),
    onError: error => console.error(error, 'ERROR'),
  });

  if (error) return <Text>{error.message}</Text>;

  return (
    <SafeAreaView style={{ height: '100%' }}>
      <View
        style={{
          height: '95%',
```

```
          display: 'flex',
          flexDirection: 'column',
          paddingHorizontal: 8,
        }}
      >
        <ScrollView style={{ flex: 1 }}>
          {messages.map(m => (
            <View key={m.id} style={{ marginVertical: 8 }}>
              <View>
                <Text style={{ fontWeight: 700 }}>{m.role}</Text>
                {m.parts.map((part, i) => {
                  switch (part.type) {
                    case 'text':
                      return (
                        <Text key={`${m.id}-${i}`}>
                          {part.text}
                        </Text>
                      );
                    case 'tool-weather':
                      return (
                        <Text key={`${m.id}-${i}`}>
                          {JSON.stringify(part, null, 2)}
                        </Text>
                      );
                  }
                })}
              </View>
            </View>
          ))}
        </ScrollView>

        <View style={{ marginTop: 8 }}>
          <TextInput
            style={{ backgroundColor: 'white', padding: 8 }}
            placeholder="Say something..."
            value={input}
            onChange={e => setInput(e.nativeEvent.text)}
            onSubmitEditing={e => {
              e.preventDefault();
              sendMessage({ text: input });
              setInput('');
            }}
            autoFocus={true}
          />
        </View>
      </View>
    </SafeAreaView>
  );
}
```

You may need to restart your development server for the changes to take effect.

With this change, you're updating the UI to handle different message parts. For text parts, you display the text content as before. For weather tool invocations, you display a JSON representation of the tool call and its result.

Now, when you ask about the weather, you'll see the tool call and its result displayed in your chat interface.

Enabling Multi-Step Tool Calls

You may have noticed that while the tool results are visible in the chat interface, the model isn't using this information to answer your original query. This is because once the model generates a tool call, it has technically completed its generation.

To solve this, you can enable multi-step tool calls using `stopWhen`. By default, `stopWhen` is set to `stepCountIs(1)`, which means generation stops after the first step when there are tool results. By changing this condition, you can allow the model to automatically send tool results back to itself to trigger additional generations until your specified stopping condition is met. In this case, you want the model to continue generating so it can use the weather tool results to answer your original question.

Update Your API Route

Modify your `app/api/chat+api.ts` file to include the `stopWhen` condition:

```
import { openai } from '@ai-sdk/openai';

import {
  streamText,
  UIMessage,
  convertToModelMessages,
  tool,
  stepCountIs,
} from 'ai';

import { z } from 'zod';

export async function POST(req: Request) {
  const { messages }: { messages: UIMessage[] } = await req.json();

  const result = streamText({
    model: openai('gpt-4o'),
    messages: convertToModelMessages(messages),
    stopWhen: stepCountIs(5),
    tools: {
      weather: tool({
        description: 'Get the weather in a location (fahrenheit)',
        inputSchema: z.object({
          location: z.string().describe('The location to get the weather for'),
        }),
        execute: async ({ location }) => {
          const temperature = Math.round(Math.random() * (90 - 32) + 32);
          return {
            location,
            temperature,
          };
        },
      }),
    },
  });
```

```
  return result.toUIMessageStreamResponse({
    headers: {
      'Content-Type': 'application/octet-stream',
      'Content-Encoding': 'none',
    },
  });
}
```

You may need to restart your development server for the changes to take effect.

Head back to the Expo app and ask about the weather in a location. You should now see the model using the weather tool results to answer your question.

By setting `stopWhen: stepCountIs(5)`, you're allowing the model to use up to 5 "steps" for any given generation. This enables more complex interactions and allows the model to gather and process information over several steps if needed. You can see this in action by adding another tool to convert the temperature from Fahrenheit to Celsius.

Add More Tools

Update your `app/api/chat+api.ts` file to add a new tool to convert the temperature from Fahrenheit to Celsius:

```
import { openai } from '@ai-sdk/openai';

import {
  streamText,
  UIMessage,
  convertToModelMessages,
  tool,
  stepCountIs,
} from 'ai';

import { z } from 'zod';

export async function POST(req: Request) {
  const { messages }: { messages: UIMessage[] } = await req.json();

  const result = streamText({
    model: openai('gpt-4o'),
    messages: convertToModelMessages(messages),
    stopWhen: stepCountIs(5),
    tools: {
      weather: tool({
        description: 'Get the weather in a location (fahrenheit)',
        inputSchema: z.object({
          location: z.string().describe('The location to get the weather for'),
        }),
        execute: async ({ location }) => {
          const temperature = Math.round(Math.random() * (90 - 32) + 32);
          return {
            location,
            temperature,
          };
```

```
        },
      }),
      convertFahrenheitToCelsius: tool({
        description: 'Convert a temperature in fahrenheit to celsius',
        inputSchema: z.object({
          temperature: z
            .number()
            .describe('The temperature in fahrenheit to convert'),
        }),
        execute: async ({ temperature }) => {
          const celsius = Math.round((temperature - 32) * (5 / 9));
          return {
            celsius,
          };
        },
      }),
    },
  });

  return result.toUIMessageStreamResponse({
    headers: {
      'Content-Type': 'application/octet-stream',
      'Content-Encoding': 'none',
    },
  });
}
```

You may need to restart your development server for the changes to take effect.

Update the UI for the new tool

To display the temperature conversion tool invocation in your UI, update your `app/(tabs)/index.tsx` file to handle the new tool part:

```
import { generateAPIUrl } from '@/utils';

import { useChat } from '@ai-sdk/react';

import { DefaultChatTransport } from 'ai';

import { fetch as expoFetch } from 'expo/fetch';

import { useState } from 'react';

import { View, TextInput, ScrollView, Text, SafeAreaView } from 'react-native';

export default function App() {
  const [input, setInput] = useState('');

  const { messages, error, sendMessage } = useChat({
    transport: new DefaultChatTransport({
      fetch: expoFetch as unknown as typeof globalThis.fetch,
      api: generateAPIUrl('/api/chat'),
    }),
    onError: error => console.error(error, 'ERROR'),
```

```
});

if (error) return <Text>{error.message}</Text>;

return (
  <SafeAreaView style={{ height: '100%' }}>
    <View
      style={{
        height: '95%',
        display: 'flex',
        flexDirection: 'column',
        paddingHorizontal: 8,
      }}
    >
      <ScrollView style={{ flex: 1 }}>
        {messages.map(m => (
          <View key={m.id} style={{ marginVertical: 8 }}>
            <View>
              <Text style={{ fontWeight: 700 }}>{m.role}</Text>
              {m.parts.map((part, i) => {
                switch (part.type) {
                  case 'text':
                    return (
                      <Text key={`${m.id}-${i}`}>
                        {part.text}
                      </Text>
                    );
                  case 'tool-weather':
                  case 'tool-convertFahrenheitToCelsius':
                    return (
                      <Text key={`${m.id}-${i}`}>
                        {JSON.stringify(part, null, 2)}
                      </Text>
                    );
                }
              })}
            </View>
          </View>
        ))}
      </ScrollView>

      <View style={{ marginTop: 8 }}>
        <TextInput
          style={{ backgroundColor: 'white', padding: 8 }}
          placeholder="Say something..."
          value={input}
          onChange={e => setInput(e.nativeEvent.text)}
          onSubmitEditing={e => {
            e.preventDefault();
            sendMessage({ text: input });
            setInput('');
          }}
          autoFocus={true}
        />
```

```
        </View>
      </View>
    </SafeAreaView>
  );
}
```

You may need to restart your development server for the changes to take effect.

Now, when you ask "What's the weather in New York in celsius?", you should see a more complete interaction:

1. The model will call the weather tool for New York.
2. You'll see the tool result displayed.
3. It will then call the temperature conversion tool to convert the temperature from Fahrenheit to Celsius.
4. The model will then use that information to provide a natural language response about the weather in New York.

This multi-step approach allows the model to gather information and use it to provide more accurate and contextual responses, making your chatbot considerably more useful.

This simple example demonstrates how tools can expand your model's capabilities. You can create more complex tools to integrate with real APIs, databases, or any other external systems, allowing the model to access and process real-world data in real-time. Tools bridge the gap between the model's knowledge cutoff and current information.

Polyfills

Several functions that are internally used by the AI SDK might not available in the Expo runtime depending on your configuration and the target platform.

First, install the following packages:

```
pnpm add @ungap/structured-clone @stardazed/streams-text-encoding
```

Then create a new file in the root of your project with the following polyfills:

```
import { Platform } from 'react-native';

import structuredClone from '@ungap/structured-clone';

if (Platform.OS !== 'web') {
  const setupPolyfills = async () => {
    const { polyfillGlobal } = await import(
      'react-native/Libraries/Utilities/PolyfillFunctions'
    );

    const { TextEncoderStream, TextDecoderStream } = await import(
      '@stardazed/streams-text-encoding'
    );

    if (!('structuredClone' in global)) {
      polyfillGlobal('structuredClone', () => structuredClone);
    }
```

```
    polyfillGlobal('TextEncoderStream', () => TextEncoderStream);
    polyfillGlobal('TextDecoderStream', () => TextDecoderStream);
  };

  setupPolyfills();
}

export {};
```

Finally, import the polyfills in your root `_layout.tsx`:

```
import '@/polyfills';
```

Where to Next?

You've built an AI chatbot using the AI SDK! From here, you have several paths to explore:

- To learn more about the AI SDK, read through the documentation.
- If you're interested in diving deeper with guides, check out the RAG (retrieval-augmented generation) and multi-modal chatbot guides.
- To jumpstart your first AI project, explore available templates.

AI SDK Core

Large Language Models (LLMs) are advanced programs that can understand, create, and engage with human language on a large scale.
They are trained on vast amounts of written material to recognize patterns in language and predict what might come next in a given piece of text.

AI SDK Core **simplifies working with LLMs by offering a standardized way of integrating them into your app** - so you can focus on building great AI applications for your users, not waste time on technical details.

For example, here's how you can generate text with various models using the AI SDK:

```
import { generateText } from "ai"
import { xai } from "@ai-sdk/xai"

const { text } = await generateText({
  model: xai("grok-3-beta"),
  prompt: "What is love?"
})
```

Love is a universal emotion that is characterized by feelings of affection, attachment, and warmth towards someone or something. It is a complex and multifaceted experience that can take many different forms, including romantic love, familial love, platonic love, and self-love.

AI SDK Core Functions

AI SDK Core has various functions designed for text generation, structured data generation, and tool usage.
These functions take a standardized approach to setting up prompts and settings, making it easier to work with different models.

- `generateText`: Generates text and tool calls.
  This function is ideal for non-interactive use cases such as automation tasks where you need to write text (e.g. drafting email or summarizing web pages) and for agents that use tools.
- `streamText`: Stream text and tool calls.
  You can use the `streamText` function for interactive use cases such as chat bots and content streaming.
- `generateObject`: Generates a typed, structured object that matches a Zod schema.
  You can use this function to force the language model to return structured data, e.g. for information extraction, synthetic data generation, or classification tasks.
- `streamObject`: Stream a structured object that matches a Zod schema.
  You can use this function to stream generated UIs.

API Reference

Please check out the AI SDK Core API Reference for more details on each function.

Generating and Streaming Text

Large language models (LLMs) can generate text in response to a prompt, which can contain instructions and information to process.
For example, you can ask a model to come up with a recipe, draft an email, or summarize a document.

The AI SDK Core provides two functions to generate text and stream it from LLMs:

- `generateText`: Generates text for a given prompt and model.
- `streamText`: Streams text from a given prompt and model.

Advanced LLM features such as tool calling and structured data generation are built on top of text generation.

`generateText`

You can generate text using the `generateText` function. This function is ideal for non-interactive use cases where you need to write text (e.g. drafting email or summarizing web pages) and for agents that use tools.

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

const { text } = await generateText({
  model: 'openai/gpt-4.1',
  prompt: 'Write a vegetarian lasagna recipe for 4 people.',
});
```

You can use more advanced prompts to generate text with more complex instructions and content:

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

const { text } = await generateText({
  model: 'openai/gpt-4.1',
  system:
    'You are a professional writer. ' +
    'You write simple, clear, and concise content.',
  prompt: `Summarize the following article in 3-5 sentences: ${article}`,
});
```

The result object of `generateText` contains several promises that resolve when all required data is available:

- `result.content`: The content that was generated in the last step.
- `result.text`: The generated text.
- `result.reasoning`: The full reasoning that the model has generated in the last step.
- `result.reasoningText`: The reasoning text of the model (only available for some models).
- `result.files`: The files that were generated in the last step.
- `result.sources`: Sources that have been used as references in the last step (only available for some models).
- `result.toolCalls`: The tool calls that were made in the last step.
- `result.toolResults`: The results of the tool calls from the last step.
- `result.finishReason`: The reason the model finished generating text.
- `result.usage`: The usage of the model during the final step of text generation.

- `result.totalUsage`: The total usage across all steps (for multi-step generations).
- `result.warnings`: Warnings from the model provider (e.g. unsupported settings).
- `result.request`: Additional request information.
- `result.response`: Additional response information, including response messages and body.
- `result.providerMetadata`: Additional provider-specific metadata.
- `result.steps`: Details for all steps, useful for getting information about intermediate steps.
- `result.experimental_output`: The generated structured output using the `experimental_output` specification.

Accessing response headers & body

Sometimes you need access to the full response from the model provider,
e.g. to access some provider-specific headers or body content.

You can access the raw response headers and body using the `response` property:

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

const result = await generateText({
  // ...
});

console.log(JSON.stringify(result.response.headers, null, 2));

console.log(JSON.stringify(result.response.body, null, 2));
```

`streamText`

Depending on your model and prompt, it can take a large language model (LLM) up to a minute to finish generating its response. This delay can be unacceptable for interactive use cases such as chatbots or real-time applications, where users expect immediate responses.

AI SDK Core provides the `streamText` function which simplifies streaming text from LLMs:

```
import { streamText } from 'ai';

const result = streamText({
  model: 'openai/gpt-4.1',
  prompt: 'Invent a new holiday and describe its traditions.',
});

// example: use textStream as an async iterable
for await (const textPart of result.textStream) {
  console.log(textPart);
}
```

`result.textStream` is both a `ReadableStream` and an `AsyncIterable`.

`streamText` immediately starts streaming and suppresses errors to prevent server crashes. Use the `onError` callback to log errors.

You can use `streamText` on its own or in combination with AI SDK UI and AI SDK RSC.
The result object contains several helper functions to make the integration into AI SDK UI easier:

- `result.toUIMessageStreamResponse()`: Creates a UI Message stream HTTP response (with tool calls etc.) that can be used in a Next.js App Router API route.
- `result.pipeUIMessageStreamToResponse()`: Writes UI Message stream delta output to a Node.js response-like object.
- `result.toTextStreamResponse()`: Creates a simple text stream HTTP response.
- `result.pipeTextStreamToResponse()`: Writes text delta output to a Node.js response-like object.

`streamText` is using backpressure and only generates tokens as they are requested. You need to consume the stream in order for it to finish.

It also provides several promises that resolve when the stream is finished:

- `result.content`: The content that was generated in the last step.
- `result.text`: The generated text.
- `result.reasoning`: The full reasoning that the model has generated.
- `result.reasoningText`: The reasoning text of the model (only available for some models).
- `result.files`: Files that have been generated by the model in the last step.
- `result.sources`: Sources that have been used as references in the last step (only available for some models).
- `result.toolCalls`: The tool calls that have been executed in the last step.
- `result.toolResults`: The tool results that have been generated in the last step.
- `result.finishReason`: The reason the model finished generating text.
- `result.usage`: The usage of the model during the final step of text generation.
- `result.totalUsage`: The total usage across all steps (for multi-step generations).
- `result.warnings`: Warnings from the model provider (e.g. unsupported settings).
- `result.steps`: Details for all steps, useful for getting information about intermediate steps.
- `result.request`: Additional request information from the last step.
- `result.response`: Additional response information from the last step.
- `result.providerMetadata`: Additional provider-specific metadata from the last step.

`onError` callback

`streamText` immediately starts streaming to enable sending data without waiting for the model. Errors become part of the stream and are not thrown to prevent e.g. servers from crashing.

To log errors, you can provide an `onError` callback that is triggered when an error occurs.

```
import { streamText } from 'ai';

const result = streamText({
  model: 'openai/gpt-4.1',
  prompt: 'Invent a new holiday and describe its traditions.',
  onError({ error }) {
    console.error(error); // your error logging logic here
  },
});
```

`onChunk` callback

When using `streamText`, you can provide an `onChunk` callback that is triggered for each chunk of the stream.

It receives the following chunk types:

- `text`
- `reasoning`
- `source`
- `tool-call`
- `tool-input-start`
- `tool-input-delta`
- `tool-result`
- `raw`

```
import { streamText } from 'ai';

const result = streamText({
  model: 'openai/gpt-4.1',
  prompt: 'Invent a new holiday and describe its traditions.',
  onChunk({ chunk }) {
    // implement your own logic here, e.g.:
    if (chunk.type === 'text') {
      console.log(chunk.text);
    }
  },
});
```

`onFinish` callback

When using `streamText`, you can provide an `onFinish` callback that is triggered when the stream is finished (API Reference).
It contains the text, usage information, finish reason, messages, steps, total usage, and more:

```
import { streamText } from 'ai';

const result = streamText({
  model: 'openai/gpt-4.1',
  prompt: 'Invent a new holiday and describe its traditions.',
  onFinish({ text, finishReason, usage, response, steps, totalUsage }) {
    // your own logic, e.g. for saving the chat history or recording usage

    const messages = response.messages; // messages that were generated
  },
});
```

`fullStream` property

You can read a stream with all events using the `fullStream` property.
This can be useful if you want to implement your own UI or handle the stream in a different way.
Here is an example of how to use the `fullStream` property:

```
import { streamText } from 'ai';

import { z } from 'zod';

const result = streamText({
  model: 'openai/gpt-4.1',
  tools: {
    cityAttractions: {
```

```
      inputSchema: z.object({ city: z.string() }),
      execute: async ({ city }) => ({
        attractions: ['attraction1', 'attraction2', 'attraction3'],
      }),
    },
  },
  prompt: 'What are some San Francisco tourist attractions?',
});

for await (const part of result.fullStream) {
  switch (part.type) {
    case 'start': {
      // handle start of stream
      break;
    }
    case 'start-step': {
      // handle start of step
      break;
    }
    case 'text-start': {
      // handle text start
      break;
    }
    case 'text-delta': {
      // handle text delta here
      break;
    }
    case 'text-end': {
      // handle text end
      break;
    }
    case 'reasoning-start': {
      // handle reasoning start
      break;
    }
    case 'reasoning-delta': {
      // handle reasoning delta here
      break;
    }
    case 'reasoning-end': {
      // handle reasoning end
      break;
    }
    case 'source': {
      // handle source here
      break;
    }
    case 'file': {
      // handle file here
      break;
    }
    case 'tool-call': {
      switch (part.toolName) {
        case 'cityAttractions': {
```

```
            // handle tool call here
            break;
          }
        }
        break;
      }
      case 'tool-input-start': {
        // handle tool input start
        break;
      }
      case 'tool-input-delta': {
        // handle tool input delta
        break;
      }
      case 'tool-input-end': {
        // handle tool input end
        break;
      }
      case 'tool-result': {
        switch (part.toolName) {
          case 'cityAttractions': {
            // handle tool result here
            break;
          }
        }
        break;
      }
      case 'tool-error': {
        // handle tool error
        break;
      }
      case 'finish-step': {
        // handle finish step
        break;
      }
      case 'finish': {
        // handle finish here
        break;
      }
      case 'error': {
        // handle error here
        break;
      }
      case 'raw': {
        // handle raw value
        break;
      }
    }
}
```

Stream transformation

You can use the `experimental_transform` option to transform the stream. This is useful for e.g. filtering, changing, or smoothing the text stream.

The transformations are applied before the callbacks are invoked and the promises are resolved.
If you e.g. have a transformation that changes all text to uppercase, the `onFinish` callback will receive the transformed text.

Smoothing streams

The AI SDK Core provides a smoothStream function that
can be used to smooth out text streaming.

```
import { smoothStream, streamText } from 'ai';

const result = streamText({
  model,
  prompt,
  experimental_transform: smoothStream(),
});
```

Custom transformations

You can also implement your own custom transformations.
The transformation function receives the tools that are available to the model,
and returns a function that is used to transform the stream.
Tools can either be generic or limited to the tools that you are using.

Here is an example of how to implement a custom transformation that converts
all text to uppercase:

```
const upperCaseTransform =
  <TOOLS extends ToolSet>() =>
  (options: { tools: TOOLS; stopStream: () => void }) =>
    new TransformStream<TextStreamPart<TOOLS>, TextStreamPart<TOOLS>>({
      transform(chunk, controller) {
        controller.enqueue(
          // for text chunks, convert the text to uppercase:
          chunk.type === 'text'
            ? { ...chunk, text: chunk.text.toUpperCase() }
            : chunk,
        );
      },
    });
```

You can also stop the stream using the `stopStream` function.
This is e.g. useful if you want to stop the stream when model guardrails are violated, e.g. by generating inappropriate content.

When you invoke `stopStream`, it is important to simulate the `step-finish` and `finish` events to guarantee that a well-formed stream is returned
and all callbacks are invoked.

```
const stopWordTransform =
  <TOOLS extends ToolSet>() =>
  ({ stopStream }: { stopStream: () => void }) =>
    new TransformStream<TextStreamPart<TOOLS>, TextStreamPart<TOOLS>>({
      // note: this is a simplified transformation for testing;
      // in a real-world version more there would need to be
      // stream buffering and scanning to correctly emit prior text
```

```
  // and to detect all STOP occurrences.
  transform(chunk, controller) {
    if (chunk.type !== 'text') {
      controller.enqueue(chunk);
      return;
    }

    if (chunk.text.includes('STOP')) {
      // stop the stream
      stopStream();

      // simulate the finish-step event
      controller.enqueue({
        type: 'finish-step',
        finishReason: 'stop',
        logprobs: undefined,
        usage: {
          completionTokens: NaN,
          promptTokens: NaN,
          totalTokens: NaN,
        },
        request: {},
        response: {
          id: 'response-id',
          modelId: 'mock-model-id',
          timestamp: new Date(0),
        },
        warnings: [],
        isContinued: false,
      });

      // simulate the finish event
      controller.enqueue({
        type: 'finish',
        finishReason: 'stop',
        logprobs: undefined,
        usage: {
          completionTokens: NaN,
          promptTokens: NaN,
          totalTokens: NaN,
        },
        response: {
          id: 'response-id',
          modelId: 'mock-model-id',
          timestamp: new Date(0),
        },
      });

      return;
    }

    controller.enqueue(chunk);
  },
});
```

Multiple transformations

You can also provide multiple transformations. They are applied in the order they are provided.

```
const result = streamText({
  model,
  prompt,
  experimental_transform: [firstTransform, secondTransform],
});
```

Sources

Some providers such as Perplexity and
Google Generative AI include sources in the response.

Currently sources are limited to web pages that ground the response.
You can access them using the `sources` property of the result.

Each `url` source contains the following properties:

- `id` : The ID of the source.
- `url` : The URL of the source.
- `title` : The optional title of the source.
- `providerMetadata` : Provider metadata for the source.

When you use `generateText` , you can access the sources using the `sources` property:

```
const result = await generateText({
  model: google('gemini-2.5-flash'),
  tools: {
    google_search: google.tools.googleSearch({}),
  },
  prompt: 'List the top 5 San Francisco news from the past week.',
});

for (const source of result.sources) {
  if (source.sourceType === 'url') {
    console.log('ID:', source.id);
    console.log('Title:', source.title);
    console.log('URL:', source.url);
    console.log('Provider metadata:', source.providerMetadata);
    console.log();
  }
}
```

When you use `streamText` , you can access the sources using the `fullStream` property:

```
const result = streamText({
  model: google('gemini-2.5-flash'),
  tools: {
    google_search: google.tools.googleSearch({}),
  },
  prompt: 'List the top 5 San Francisco news from the past week.',
});

for await (const part of result.fullStream) {
```

```
  if (part.type === 'source' && part.sourceType === 'url') {
    console.log('ID:', part.id);
    console.log('Title:', part.title);
    console.log('URL:', part.url);
    console.log('Provider metadata:', part.providerMetadata);
    console.log();
  }
}
```

The sources are also available in the `result.sources` promise.

Examples

You can see `generateText` and `streamText` in action using various frameworks in the following examples:

`generateText`

Learn to generate text in Node.js
Learn to generate text in Next.js with Route Handlers (AI SDK UI)
Learn to generate text in Next.js with Server Actions (AI SDK RSC)

`streamText`

Learn to stream text in Node.js
Learn to stream text in Next.js with Route Handlers (AI SDK UI)
Learn to stream text in Next.js with Server Actions (AI SDK RSC)

< Previous: Overview | Next: Generating Structured Data

Language Model Middleware

Language model middleware is a way to enhance the behavior of language models by intercepting and modifying the calls to the language model.

It can be used to add features like guardrails, RAG, caching, and logging in a language model agnostic way. Such middleware can be developed and distributed independently from the language models that they are applied to.

Using Language Model Middleware

You can use language model middleware with the `wrapLanguageModel` function. It takes a language model and a language model middleware and returns a new language model that incorporates the middleware.

```
import { wrapLanguageModel } from 'ai';

const wrappedLanguageModel = wrapLanguageModel({
  model: yourModel,
  middleware: yourLanguageModelMiddleware,
});
```

The wrapped language model can be used just like any other language model, e.g. in `streamText`:

```
const result = streamText({
  model: wrappedLanguageModel,
  prompt: 'What cities are in the United States?',
});
```

Multiple middlewares

You can provide multiple middlewares to the `wrapLanguageModel` function. The middlewares will be applied in the order they are provided.

```
const wrappedLanguageModel = wrapLanguageModel({
  model: yourModel,
  middleware: [firstMiddleware, secondMiddleware],
});

// applied as: firstMiddleware(secondMiddleware(yourModel))
```

Built-in Middleware

The AI SDK comes with several built-in middlewares that you can use to configure language models:

- `extractReasoningMiddleware`: Extracts reasoning information from the generated text and exposes it as a `reasoning` property on the result.
- `simulateStreamingMiddleware`: Simulates streaming behavior with responses from non-streaming language models.
- `defaultSettingsMiddleware`: Applies default settings to a language model.

Extract Reasoning

Some providers and models expose reasoning information in the generated text using special tags, e.g. `<think>` and `</think>`.

The `extractReasoningMiddleware` function can be used to extract this reasoning information and expose it as a `reasoning` property on the result.

```
import { wrapLanguageModel, extractReasoningMiddleware } from 'ai';

const model = wrapLanguageModel({
  model: yourModel,
  middleware: extractReasoningMiddleware({ tagName: 'think' }),
});
```

You can then use that enhanced model in functions like `generateText` and `streamText`.

The `extractReasoningMiddleware` function also includes a `startWithReasoning` option. When set to `true`, the reasoning tag will be prepended to the generated text. This is useful for models that do not include the reasoning tag at the beginning of the response. For more details, see the DeepSeek R1 guide.

Simulate Streaming

The `simulateStreamingMiddleware` function can be used to simulate streaming behavior with responses from non-streaming language models. This is useful when you want to maintain a consistent streaming interface even when using models that only provide complete responses.

```
import { wrapLanguageModel, simulateStreamingMiddleware } from 'ai';

const model = wrapLanguageModel({
  model: yourModel,
  middleware: simulateStreamingMiddleware(),
});
```

Default Settings

The `defaultSettingsMiddleware` function can be used to apply default settings to a language model.

```
import { wrapLanguageModel, defaultSettingsMiddleware } from 'ai';

const model = wrapLanguageModel({
  model: yourModel,
  middleware: defaultSettingsMiddleware({
    settings: {
      temperature: 0.5,
      maxOutputTokens: 800,
      providerOptions: { openai: { store: false } },
    },
  }),
});
```

Community Middleware

The AI SDK provides a Language Model Middleware specification. Community members can develop middleware that adheres to this specification, making it compatible with the AI SDK ecosystem.

Here are some community middlewares that you can explore:

Custom tool call parser

The Custom tool call parser middleware extends tool call capabilities to models that don't natively support the OpenAI-style `tools` parameter. This includes many self-hosted and third-party models that lack native function calling features.

Using this middleware on models that support native function calls may result in unintended performance degradation, so check whether your model supports native function calls before deciding to use it.

This middleware enables function calling capabilities by converting function schemas into prompt instructions and parsing the model's responses into structured function calls. It works by transforming the JSON function definitions into natural language instructions the model can understand, then analyzing the generated text to extract function call attempts. This approach allows developers to use the same function calling API across different model providers, even with models that don't natively support the OpenAI-style function calling format, providing a consistent function calling experience regardless of the underlying model implementation.

The `@ai-sdk-tool/parser` package offers three middleware variants:

- `createToolMiddleware`: A flexible function for creating custom tool call middleware tailored to specific models
- `hermesToolMiddleware`: Ready-to-use middleware for Hermes & Qwen format function calls
- `gemmaToolMiddleware`: Pre-configured middleware for Gemma 3 model series function call format

Here's how you can enable function calls with Gemma models that don't support them natively:

```
import { wrapLanguageModel } from 'ai';
import { gemmaToolMiddleware } from '@ai-sdk-tool/parser';

const model = wrapLanguageModel({
  model: openrouter('google/gemma-3-27b-it'),
  middleware: gemmaToolMiddleware,
});
```

Find more examples at this link.

Implementing Language Model Middleware

Implementing language model middleware is advanced functionality and requires a solid understanding of the language model specification.

You can implement any of the following three functions to modify the behavior of the language model:

1. `transformParams`: Transforms the parameters before they are passed to the language model, for both `doGenerate` and `doStream`.
2. `wrapGenerate`: Wraps the `doGenerate` method of the language model. You can modify the parameters, call the language model, and modify the result.
3. `wrapStream`: Wraps the `doStream` method of the language model. You can modify the parameters, call the language model, and modify the result.

Here are some examples of how to implement language model middleware:

Examples

These examples are not meant to be used in production. They are just to show how you can use middleware to enhance the behavior of language models.

Logging

This example shows how to log the parameters and generated text of a language model call.

```
import type {
  LanguageModelV2Middleware,
  LanguageModelV2StreamPart,
} from '@ai-sdk/provider';

export const yourLogMiddleware: LanguageModelV2Middleware = {
  wrapGenerate: async ({ doGenerate, params }) => {
    console.log('doGenerate called');
    console.log(`params: ${JSON.stringify(params, null, 2)}`);

    const result = await doGenerate();

    console.log('doGenerate finished');
    console.log(`generated text: ${result.text}`);

    return result;
  },

  wrapStream: async ({ doStream, params }) => {
    console.log('doStream called');
    console.log(`params: ${JSON.stringify(params, null, 2)}`);

    const { stream, ...rest } = await doStream();

    let generatedText = '';
    const textBlocks = new Map<string, string>();

    const transformStream = new TransformStream<
      LanguageModelV2StreamPart,
      LanguageModelV2StreamPart
    >({
      transform(chunk, controller) {
        switch (chunk.type) {
          case 'text-start': {
            textBlocks.set(chunk.id, '');
            break;
          }
          case 'text-delta': {
            const existing = textBlocks.get(chunk.id) || '';
            textBlocks.set(chunk.id, existing + chunk.delta);
            generatedText += chunk.delta;
            break;
          }
          case 'text-end': {
            console.log(
```

```
              `Text block ${chunk.id} completed:`,
              textBlocks.get(chunk.id),
            );
            break;
          }
        }

        controller.enqueue(chunk);
      },

      flush() {
        console.log('doStream finished');
        console.log(`generated text: ${generatedText}`);
      },
    });

    return {
      stream: stream.pipeThrough(transformStream),
      ...rest,
    };
  },
};
```

Caching

This example shows how to build a simple cache for the generated text of a language model call.

```
import type { LanguageModelV2Middleware } from '@ai-sdk/provider';

const cache = new Map<string, any>();

export const yourCacheMiddleware: LanguageModelV2Middleware = {
  wrapGenerate: async ({ doGenerate, params }) => {
    const cacheKey = JSON.stringify(params);

    if (cache.has(cacheKey)) {
      return cache.get(cacheKey);
    }

    const result = await doGenerate();

    cache.set(cacheKey, result);

    return result;
  },

  // here you would implement the caching logic for streaming
};
```

Retrieval Augmented Generation (RAG)

This example shows how to use RAG as middleware.

Helper functions like `getLastUserMessageText` and `findSources` are not part of the AI SDK. They are just used in this example to illustrate the concept of RAG.

```
import type { LanguageModelV2Middleware } from '@ai-sdk/provider';

export const yourRagMiddleware: LanguageModelV2Middleware = {
  transformParams: async ({ params }) => {
    const lastUserMessageText = getLastUserMessageText({
      prompt: params.prompt,
    });

    if (lastUserMessageText == null) {
      return params; // do not use RAG (send unmodified parameters)
    }

    const instruction =
      'Use the following information to answer the question:\n' +
      findSources({ text: lastUserMessageText })
        .map(chunk => JSON.stringify(chunk))
        .join('\n');

    return addToLastUserMessage({ params, text: instruction });
  },
};
```

Guardrails

Guard rails are a way to ensure that the generated text of a language model call is safe and appropriate. This example shows how to use guardrails as middleware.

```
import type { LanguageModelV2Middleware } from '@ai-sdk/provider';

export const yourGuardrailMiddleware: LanguageModelV2Middleware = {
  wrapGenerate: async ({ doGenerate }) => {
    const { text, ...rest } = await doGenerate();

    // filtering approach, e.g. for PII or other sensitive information:
    const cleanedText = text?.replace(/badword/g, '<REDACTED>');

    return { text: cleanedText, ...rest };
  },

  // here you would implement the guardrail logic for streaming
  // Note: streaming guardrails are difficult to implement, because
  // you do not know the full content of the stream until it's finished.
};
```

Configuring Per Request Custom Metadata

To send and access custom metadata in Middleware, you can use `providerOptions`. This is useful when building logging middleware where you want to pass additional context like user IDs, timestamps, or other contextual data that can help with tracking and debugging.

```
import { openai } from '@ai-sdk/openai';

import { generateText, wrapLanguageModel } from 'ai';
```

```
import type { LanguageModelV2Middleware } from '@ai-sdk/provider';

export const yourLogMiddleware: LanguageModelV2Middleware = {
  wrapGenerate: async ({ doGenerate, params }) => {
    console.log('METADATA', params?.providerMetadata?.yourLogMiddleware);

    const result = await doGenerate();

    return result;
  },
};

const { text } = await generateText({
  model: wrapLanguageModel({
    model: openai('gpt-4o'),
    middleware: yourLogMiddleware,
  }),
  prompt: 'Invent a new holiday and describe its traditions.',
  providerOptions: {
    yourLogMiddleware: {
      hello: 'world',
    },
  },
});

console.log(text);
```

Provider & Model Management

When you work with multiple providers and models, it is often desirable to manage them in a central place and access the models through simple string ids.

The AI SDK offers custom providers and a provider registry for this purpose:

- With **custom providers**, you can pre-configure model settings, provide model name aliases, and limit the available models.
- The **provider registry** lets you mix multiple providers and access them through simple string ids.

You can mix and match custom providers, the provider registry, and middleware in your application.

Custom Providers

You can create a custom provider using `customProvider`.

Example: custom model settings

You might want to override the default model settings for a provider or provide model name aliases with pre-configured settings.

```
import { openai as originalOpenAI } from '@ai-sdk/openai';
import {
  customProvider,
  defaultSettingsMiddleware,
  wrapLanguageModel,
} from 'ai';

// custom provider with different provider options:
export const openai = customProvider({
  languageModels: {
    // replacement model with custom provider options:
    'gpt-4o': wrapLanguageModel({
      model: originalOpenAI('gpt-4o'),
      middleware: defaultSettingsMiddleware({
        settings: {
          providerOptions: {
            openai: {
              reasoningEffort: 'high',
            },
          },
        },
      }),
    }),

    // alias model with custom provider options:
    'gpt-4o-mini-high-reasoning': wrapLanguageModel({
      model: originalOpenAI('gpt-4o-mini'),
      middleware: defaultSettingsMiddleware({
        settings: {
          providerOptions: {
            openai: {
```

```
              reasoningEffort: 'high',
            },
          },
        },
      }),
    }),
  },
  fallbackProvider: originalOpenAI,
});
```

Example: model name alias

You can also provide model name aliases, so you can update the model version in one place in the future:

```
import { anthropic as originalAnthropic } from '@ai-sdk/anthropic';
import { customProvider } from 'ai';

// custom provider with alias names:
export const anthropic = customProvider({
  languageModels: {
    opus: originalAnthropic('claude-3-opus-20240229'),
    sonnet: originalAnthropic('claude-3-5-sonnet-20240620'),
    haiku: originalAnthropic('claude-3-haiku-20240307'),
  },
  fallbackProvider: originalAnthropic,
});
```

Example: limit available models

You can limit the available models in the system, even if you have multiple providers.

```
import { anthropic } from '@ai-sdk/anthropic';
import { openai } from '@ai-sdk/openai';
import {
  customProvider,
  defaultSettingsMiddleware,
  wrapLanguageModel,
} from 'ai';

export const myProvider = customProvider({
  languageModels: {
    'text-medium': anthropic('claude-3-5-sonnet-20240620'),
    'text-small': openai('gpt-4o-mini'),
    'reasoning-medium': wrapLanguageModel({
      model: openai('gpt-4o'),
      middleware: defaultSettingsMiddleware({
        settings: {
          providerOptions: {
            openai: {
              reasoningEffort: 'high',
            },
          },
        },
      }),
```

```
    }),
    'reasoning-fast': wrapLanguageModel({
      model: openai('gpt-4o-mini'),
      middleware: defaultSettingsMiddleware({
        settings: {
          providerOptions: {
            openai: {
              reasoningEffort: 'high',
            },
          },
        },
      }),
    }),
  },
  embeddingModels: {
    embedding: openai.textEmbeddingModel('text-embedding-3-small'),
  },
  // no fallback provider
});
```

Provider Registry

You can create a provider registry with multiple providers and models using `createProviderRegistry`.

Setup

```
// registry.ts
import { anthropic } from '@ai-sdk/anthropic';
import { createOpenAI } from '@ai-sdk/openai';
import { createProviderRegistry } from 'ai';

export const registry = createProviderRegistry({
  // register provider with prefix and default setup:
  anthropic,

  // register provider with prefix and custom setup:
  openai: createOpenAI({
    apiKey: process.env.OPENAI_API_KEY,
  }),
});
```

Setup with Custom Separator

By default, the registry uses `:` as the separator between provider and model IDs. You can customize this separator:

```
// registry.ts
import { createProviderRegistry } from 'ai';
import { anthropic } from '@ai-sdk/anthropic';
import { openai } from '@ai-sdk/openai';

export const customSeparatorRegistry = createProviderRegistry(
  {
```

```
    anthropic,
    openai,
  },
  { separator: ' > ' }
);
```

Example: Use language models

You can access language models by using the `languageModel` method on the registry. The provider id will become the prefix of the model id: `providerId:modelId`.

```
import { generateText } from 'ai';
import { registry } from './registry';

const { text } = await generateText({
  model: registry.languageModel('openai:gpt-4.1'), // default separator

  // or with custom separator:
  // model: customSeparatorRegistry.languageModel('openai > gpt-4.1'),
  prompt: 'Invent a new holiday and describe its traditions.',
});
```

Example: Use text embedding models

You can access text embedding models by using the `textEmbeddingModel` method on the registry. The provider id will become the prefix of the model id: `providerId:modelId`.

```
import { embed } from 'ai';
import { registry } from './registry';

const { embedding } = await embed({
  model: registry.textEmbeddingModel('openai:text-embedding-3-small'),
  value: 'sunny day at the beach',
});
```

Example: Use image models

You can access image models by using the `imageModel` method on the registry. The provider id will become the prefix of the model id: `providerId:modelId`.

```
import { generateImage } from 'ai';
import { registry } from './registry';

const { image } = await generateImage({
  model: registry.imageModel('openai:dall-e-3'),
  prompt: 'A beautiful sunset over a calm ocean',
});
```

Combining Custom Providers, Provider Registry, and Middleware

The central idea of provider management is to set up a file that contains all the providers and models you want to use. You may want to pre-configure model settings, provide model name aliases, limit the

available models, and more.

Here is an example that implements the following concepts:

- pass through a full provider with a namespace prefix (here: `xai > *`)
- setup an OpenAI-compatible provider with custom api key and base URL (here: `custom > *`)
- setup model name aliases (here: `anthropic > fast`, `anthropic > writing`, `anthropic > reasoning`)
- pre-configure model settings (here: `anthropic > reasoning`)
- validate the provider-specific options (here: `AnthropicProviderOptions`)
- use a fallback provider (here: `anthropic > *`)
- limit a provider to certain models without a fallback (here: `groq > gemma2-9b-it`, `groq > qwen-qwq-32b`)
- define a custom separator for the provider registry (here: `>`)

```
import { anthropic, AnthropicProviderOptions } from '@ai-sdk/anthropic';
import { createOpenAICompatible } from '@ai-sdk/openai-compatible';
import { xai } from '@ai-sdk/xai';
import { groq } from '@ai-sdk/groq';
import {
  createProviderRegistry,
  customProvider,
  defaultSettingsMiddleware,
  wrapLanguageModel,
} from 'ai';

export const registry = createProviderRegistry(
  {
    // pass through a full provider with a namespace prefix
    xai,

    // access an OpenAI-compatible provider with custom setup
    custom: createOpenAICompatible({
      name: 'provider-name',
      apiKey: process.env.CUSTOM_API_KEY,
      baseURL: 'https://api.custom.com/v1',
    }),

    // setup model name aliases
    anthropic: customProvider({
      languageModels: {
        fast: anthropic('claude-3-haiku-20240307'),

        // simple model
        writing: anthropic('claude-3-7-sonnet-20250219'),

        // extended reasoning model configuration:
        reasoning: wrapLanguageModel({
          model: anthropic('claude-3-7-sonnet-20250219'),
          middleware: defaultSettingsMiddleware({
            settings: {
              maxOutputTokens: 100000, // example default setting
              providerOptions: {
                anthropic: {
                  thinking: {
```

```
                    type: 'enabled',
                    budgetTokens: 32000,
                  },
                } satisfies AnthropicProviderOptions,
              },
            },
          }),
        }),
      },
      fallbackProvider: anthropic,
    }),

    // limit a provider to certain models without a fallback
    groq: customProvider({
      languageModels: {
        'gemma2-9b-it': groq('gemma2-9b-it'),
        'qwen-qwq-32b': groq('qwen-qwq-32b'),
      },
    }),
  },
  { separator: ' > ' }
);

// usage:
const model = registry.languageModel('anthropic > reasoning');
```

Global Provider Configuration

The AI SDK 5 includes a global provider feature that allows you to specify a model using just a plain model ID string:

```
import { streamText } from 'ai';

const result = await streamText({
  model: 'openai/gpt-4o', // Uses the global provider (defaults to AI Gateway)
  prompt: 'Invent a new holiday and describe its traditions.',
});
```

By default, the global provider is set to the Vercel AI Gateway.

Customizing the Global Provider

You can set your own preferred global provider:

```
// setup.ts
import { openai } from '@ai-sdk/openai';

// Initialize once during startup:
globalThis.AI_SDK_DEFAULT_PROVIDER = openai;
```

```
// app.ts
import { streamText } from 'ai';

const result = await streamText({
  model: 'gpt-4o', // Uses OpenAI provider without prefix
```

```
  prompt: 'Invent a new holiday and describe its traditions.',
});
```

This simplifies provider usage and makes it easier to switch between providers without changing your model references throughout your codebase.

Error Handling

Handling regular errors

Regular errors are thrown and can be handled using the `try/catch` block.

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

try {
  const { text } = await generateText({
    model: 'openai/gpt-4.1',
    prompt: 'Write a vegetarian lasagna recipe for 4 people.',
  });
} catch (error) {
  // handle error
}
```

See Error Types for more information on the different types of errors that may be thrown.

Handling streaming errors (simple streams)

When errors occur during streams that do not support error chunks, the error is thrown as a regular error. You can handle these errors using the `try/catch` block.

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

try {
  const { textStream } = streamText({
    model: 'openai/gpt-4.1',
    prompt: 'Write a vegetarian lasagna recipe for 4 people.',
  });

  for await (const textPart of textStream) {
    process.stdout.write(textPart);
  }
} catch (error) {
  // handle error
}
```

Handling streaming errors (streaming with `error` support)

Full streams support error parts. You can handle those parts similar to other parts. It is recommended to also add a try-catch block for errors that happen outside of the streaming.

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

try {
  const { fullStream } = streamText({
    model: 'openai/gpt-4.1',
    prompt: 'Write a vegetarian lasagna recipe for 4 people.',
```

```
  });

  for await (const part of fullStream) {
    switch (part.type) {
      // ... handle other part types

      case 'error': {
        const error = part.error;
        // handle error
        break;
      }

      case 'abort': {
        // handle stream abort
        break;
      }

      case 'tool-error': {
        const error = part.error;
        // handle error
        break;
      }
    }
  }
} catch (error) {
  // handle error
}
```

Handling stream aborts

When streams are aborted (e.g., via chat stop button), you may want to perform cleanup operations like updating stored messages in your UI. Use the `onAbort` callback to handle these cases.

The `onAbort` callback is called when a stream is aborted via `AbortSignal`, but `onFinish` is not called. This ensures you can still update your UI state appropriately.

```
import { streamText } from 'ai';

const { textStream } = streamText({
  model: 'openai/gpt-4.1',
  prompt: 'Write a vegetarian lasagna recipe for 4 people.',
  onAbort: ({ steps }) => {
    // Update stored messages or perform cleanup
    console.log('Stream aborted after', steps.length, 'steps');
  },
  onFinish: ({ steps, totalUsage }) => {
    // This is called on normal completion
    console.log('Stream completed normally');
  },
});

for await (const textPart of textStream) {
  process.stdout.write(textPart);
}
```

The `onAbort` callback receives:

- `steps`: An array of all completed steps before the abort

You can also handle abort events directly in the stream:

```
import { streamText } from 'ai';

const { fullStream } = streamText({
  model: 'openai/gpt-4.1',
  prompt: 'Write a vegetarian lasagna recipe for 4 people.',
});

for await (const chunk of fullStream) {
  switch (chunk.type) {
    case 'abort': {
      // Handle abort directly in stream
      console.log('Stream was aborted');
      break;
    }
    // ... handle other part types
  }
}
```

Testing

Testing language models can be challenging, because they are non-deterministic and calling them is slow and expensive.

To enable you to unit test your code that uses the AI SDK, the AI SDK Core includes mock providers and test helpers. You can import the following helpers from `ai/test`:

- `MockEmbeddingModelV2`: A mock embedding model using the embedding model v2 specification.
- `MockLanguageModelV2`: A mock language model using the language model v2 specification.
- `mockId`: Provides an incrementing integer ID.
- `mockValues`: Iterates over an array of values with each call. Returns the last value when the array is exhausted.
- `simulateReadableStream`: Simulates a readable stream with delays.

With mock providers and test helpers, you can control the output of the AI SDK and test your code in a repeatable and deterministic way without actually calling a language model provider.

Examples

You can use the test helpers with the AI Core functions in your unit tests:

generateText

```
import { generateText } from 'ai';
import { MockLanguageModelV2 } from 'ai/test';

const result = await generateText({
  model: new MockLanguageModelV2({
    doGenerate: async () => ({
      finishReason: 'stop',
      usage: { inputTokens: 10, outputTokens: 20, totalTokens: 30 },
      content: [{ type: 'text', text: `Hello, world!` }],
      warnings: [],
    }),
  }),
  prompt: 'Hello, test!',
});
```

streamText

```
import { streamText, simulateReadableStream } from 'ai';
import { MockLanguageModelV2 } from 'ai/test';

const result = streamText({
  model: new MockLanguageModelV2({
    doStream: async () => ({
      stream: simulateReadableStream({
        chunks: [
          { type: 'text-start', id: 'text-1' },
          { type: 'text-delta', id: 'text-1', delta: 'Hello' },
```

```
          { type: 'text-delta', id: 'text-1', delta: ', ' },
          { type: 'text-delta', id: 'text-1', delta: 'world!' },
          { type: 'text-end', id: 'text-1' },
          { type: 'finish', finishReason: 'stop', logprobs: undefined, usage: { i
        ],
      }),
    }),
  }),
  prompt: 'Hello, test!',
});
```

generateObject

```
import { generateObject } from 'ai';
import { MockLanguageModelV2 } from 'ai/test';
import { z } from 'zod';

const result = await generateObject({
  model: new MockLanguageModelV2({
    doGenerate: async () => ({
      finishReason: 'stop',
      usage: { inputTokens: 10, outputTokens: 20, totalTokens: 30 },
      content: [{ type: 'text', text: `{"content":"Hello, world!"}` }],
      warnings: [],
    }),
  }),
  schema: z.object({ content: z.string() }),
  prompt: 'Hello, test!',
});
```

streamObject

```
import { streamObject, simulateReadableStream } from 'ai';
import { MockLanguageModelV2 } from 'ai/test';
import { z } from 'zod';

const result = streamObject({
  model: new MockLanguageModelV2({
    doStream: async () => ({
      stream: simulateReadableStream({
        chunks: [
          { type: 'text-start', id: 'text-1' },
          { type: 'text-delta', id: 'text-1', delta: '{ ' },
          { type: 'text-delta', id: 'text-1', delta: '"content": ' },
          { type: 'text-delta', id: 'text-1', delta: `"Hello, ` },
          { type: 'text-delta', id: 'text-1', delta: `world` },
          { type: 'text-delta', id: 'text-1', delta: `!"` },
          { type: 'text-delta', id: 'text-1', delta: ' }' },
          { type: 'text-end', id: 'text-1' },
          { type: 'finish', finishReason: 'stop', logprobs: undefined, usage: { i
        ],
      }),
    }),
  }),
```

```
  schema: z.object({ content: z.string() }),
  prompt: 'Hello, test!',
});
```

Simulate UI Message Stream Responses

You can also simulate UI Message Stream responses for testing, debugging, or demonstration purposes.

Here is a Next example:

```
import { simulateReadableStream } from 'ai';

export async function POST(req: Request) {
  return new Response(
    simulateReadableStream({
      initialDelayInMs: 1000, // Delay before the first chunk
      chunkDelayInMs: 300,    // Delay between chunks
      chunks: [
        `data: {"type":"start","messageId":"msg-123"}\n\n`,
        `data: {"type":"text-start","id":"text-1"}\n\n`,
        `data: {"type":"text-delta","id":"text-1","delta":"This"}\n\n`,
        `data: {"type":"text-delta","id":"text-1","delta":" is an"}\n\n`,
        `data: {"type":"text-delta","id":"text-1","delta":" example."}\n\n`,
        `data: {"type":"text-end","id":"text-1"}\n\n`,
        `data: {"type":"finish"}\n\n`,
        `data: [DONE]\n\n`,
      ],
    }).pipeThrough(new TextEncoderStream()),
    {
      status: 200,
      headers: {
        'Content-Type': 'text/event-stream',
        'Cache-Control': 'no-cache',
        Connection: 'keep-alive',
        'x-vercel-ai-ui-message-stream': 'v1',
      },
    }
  );
}
```

Telemetry

AI SDK Telemetry is experimental and may change in the future.

The AI SDK uses OpenTelemetry to collect telemetry data. OpenTelemetry is an open-source observability framework designed to provide standardized instrumentation for collecting telemetry data.

Check out the AI SDK Observability Integrations to see providers that offer monitoring and tracing for AI SDK applications.

Enabling telemetry

For Next.js applications, please follow the Next.js OpenTelemetry guide to enable telemetry first.

You can then use the `experimental_telemetry` option to enable telemetry on specific function calls while the feature is experimental:

```
const result = await generateText({
  model: openai('gpt-4.1'),
  prompt: 'Write a short story about a cat.',
  experimental_telemetry: { isEnabled: true },
});
```

When telemetry is enabled, you can also control if you want to record the input values and the output values for the function. By default, both are enabled. You can disable them by setting the `recordInputs` and `recordOutputs` options to `false`.

Disabling the recording of inputs and outputs can be useful for privacy, data transfer, and performance reasons. You might for example want to disable recording inputs if they contain sensitive information.

Telemetry Metadata

You can provide a `functionId` to identify the function that the telemetry data is for, and `metadata` to include additional information in the telemetry data.

```
const result = await generateText({
  model: openai('gpt-4.1'),
  prompt: 'Write a short story about a cat.',
  experimental_telemetry: {
    isEnabled: true,
    functionId: 'my-awesome-function',
    metadata: {
      something: 'custom',
      someOtherThing: 'other-value',
    },
  },
});
```

Custom Tracer

You may provide a `tracer` which must return an OpenTelemetry `Tracer`. This is useful in situations where you want your traces to use a `TracerProvider` other than the one provided by the `@opentelemetry/api` singleton.

```
const tracerProvider = new NodeTracerProvider();

const result = await generateText({
  model: openai('gpt-4.1'),
  prompt: 'Write a short story about a cat.',
  experimental_telemetry: {
    isEnabled: true,
    tracer: tracerProvider.getTracer('ai'),
  },
});
```

Collected Data

generateText function

`generateText` records 3 types of spans:

- `ai.generateText` (span): the full length of the generateText call. It contains 1 or more `ai.generateText.doGenerate` spans. It contains the basic LLM span information and the following attributes:
  - `operation.name`: `ai.generateText` and the functionId that was set through `telemetry.functionId`
  - `ai.operationId`: `"ai.generateText"`
  - `ai.prompt`: the prompt that was used when calling `generateText`
  - `ai.response.text`: the text that was generated
  - `ai.response.toolCalls`: the tool calls that were made as part of the generation (stringified JSON)
  - `ai.response.finishReason`: the reason why the generation finished
  - `ai.settings.maxOutputTokens`: the maximum number of output tokens that were set
- `ai.generateText.doGenerate` (span): a provider doGenerate call. It can contain `ai.toolCall` spans. It contains the call LLM span information and the following attributes:
  - `operation.name`: `ai.generateText.doGenerate` and the functionId that was set through `telemetry.functionId`
  - `ai.operationId`: `"ai.generateText.doGenerate"`
  - `ai.prompt.messages`: the messages that were passed into the provider
  - `ai.prompt.tools`: array of stringified tool definitions. The tools can be of type `function` or `provider-defined-client`. Function tools have a `name`, `description` (optional), and `inputSchema` (JSON schema). Provider-defined-client tools have a `name`, `id`, and `input` (Record).
  - `ai.prompt.toolChoice`: the stringified tool choice setting (JSON). It has a `type` property (`auto`, `none`, `required`, `tool`), and if the type is `tool`, a `toolName` property with the specific tool.
  - `ai.response.text`: the text that was generated
  - `ai.response.toolCalls`: the tool calls that were made as part of the generation (stringified JSON)
  - `ai.response.finishReason`: the reason why the generation finished

- `ai.toolCall` (span): a tool call that is made as part of the generateText call. See Tool call spans for more details.

streamText function

`streamText` records 3 types of spans and 2 types of events:

- `ai.streamText` (span): the full length of the streamText call. It contains a `ai.streamText.doStream` span. It contains the basic LLM span information and the following attributes:
  - `operation.name`: `ai.streamText` and the functionId that was set through `telemetry.functionId`
  - `ai.operationId`: `"ai.streamText"`
  - `ai.prompt`: the prompt that was used when calling `streamText`
  - `ai.response.text`: the text that was generated
  - `ai.response.toolCalls`: the tool calls that were made as part of the generation (stringified JSON)
  - `ai.response.finishReason`: the reason why the generation finished
  - `ai.settings.maxOutputTokens`: the maximum number of output tokens that were set
- `ai.streamText.doStream` (span): a provider doStream call. This span contains an `ai.stream.firstChunk` event and `ai.toolCall` spans. It contains the call LLM span information and the following attributes:
  - `operation.name`: `ai.streamText.doStream` and the functionId that was set through `telemetry.functionId`
  - `ai.operationId`: `"ai.streamText.doStream"`
  - `ai.prompt.messages`: the messages that were passed into the provider
  - `ai.prompt.tools`: array of stringified tool definitions. The tools can be of type `function` or `provider-defined-client`. Function tools have a `name`, `description` (optional), and `inputSchema` (JSON schema). Provider-defined-client tools have a `name`, `id`, and `input` (Record).
  - `ai.prompt.toolChoice`: the stringified tool choice setting (JSON). It has a `type` property (`auto`, `none`, `required`, `tool`), and if the type is `tool`, a `toolName` property with the specific tool.
  - `ai.response.text`: the text that was generated
  - `ai.response.toolCalls`: the tool calls that were made as part of the generation (stringified JSON)
  - `ai.response.msToFirstChunk`: the time it took to receive the first chunk in milliseconds
  - `ai.response.msToFinish`: the time it took to receive the finish part of the LLM stream in milliseconds
  - `ai.response.avgCompletionTokensPerSecond`: the average number of completion tokens per second
  - `ai.response.finishReason`: the reason why the generation finished
- `ai.toolCall` (span): a tool call that is made as part of the generateText call. See Tool call spans for more details.
- `ai.stream.firstChunk` (event): an event that is emitted when the first chunk of the stream is received.
  - `ai.response.msToFirstChunk`: the time it took to receive the first chunk

- `ai.stream.finish` (event): an event that is emitted when the finish part of the LLM stream is received.

It also records a `ai.stream.firstChunk` event when the first chunk of the stream is received.

generateObject function

`generateObject` records 2 types of spans:

- `ai.generateObject` (span): the full length of the generateObject call. It contains 1 or more `ai.generateObject.doGenerate` spans. It contains the basic LLM span information and the following attributes:
  - `operation.name`: `ai.generateObject` and the functionId that was set through `telemetry.functionId`
  - `ai.operationId`: `"ai.generateObject"`
  - `ai.prompt`: the prompt that was used when calling `generateObject`
  - `ai.schema`: Stringified JSON schema version of the schema that was passed into the `generateObject` function
  - `ai.schema.name`: the name of the schema that was passed into the `generateObject` function
  - `ai.schema.description`: the description of the schema that was passed into the `generateObject` function
  - `ai.response.object`: the object that was generated (stringified JSON)
  - `ai.settings.output`: the output type that was used, e.g. `object` or `no-schema`
- `ai.generateObject.doGenerate` (span): a provider doGenerate call. It contains the call LLM span information and the following attributes:
  - `operation.name`: `ai.generateObject.doGenerate` and the functionId that was set through `telemetry.functionId`
  - `ai.operationId`: `"ai.generateObject.doGenerate"`
  - `ai.prompt.messages`: the messages that were passed into the provider
  - `ai.response.object`: the object that was generated (stringified JSON)
  - `ai.response.finishReason`: the reason why the generation finished

streamObject function

`streamObject` records 2 types of spans and 1 type of event:

- `ai.streamObject` (span): the full length of the streamObject call. It contains 1 or more `ai.streamObject.doStream` spans. It contains the basic LLM span information and the following attributes:
  - `operation.name`: `ai.streamObject` and the functionId that was set through `telemetry.functionId`
  - `ai.operationId`: `"ai.streamObject"`
  - `ai.prompt`: the prompt that was used when calling `streamObject`
  - `ai.schema`: Stringified JSON schema version of the schema that was passed into the `streamObject` function
  - `ai.schema.name`: the name of the schema that was passed into the `streamObject` function
  - `ai.schema.description`: the description of the schema that was passed into the `streamObject` function
  - `ai.response.object`: the object that was generated (stringified JSON)

    - `ai.settings.output`: the output type that was used, e.g. `object` or `no-schema`

- `ai.streamObject.doStream` (span): a provider doStream call. This span contains an `ai.stream.firstChunk` event. It contains the call LLM span information and the following attributes:

    - `operation.name`: `ai.streamObject.doStream` and the functionId that was set through `telemetry.functionId`
    - `ai.operationId`: `"ai.streamObject.doStream"`
    - `ai.prompt.messages`: the messages that were passed into the provider
    - `ai.response.object`: the object that was generated (stringified JSON)
    - `ai.response.msToFirstChunk`: the time it took to receive the first chunk
    - `ai.response.finishReason`: the reason why the generation finished

- `ai.stream.firstChunk` (event): an event that is emitted when the first chunk of the stream is received.

    - `ai.response.msToFirstChunk`: the time it took to receive the first chunk

embed function

`embed` records 2 types of spans:

- `ai.embed` (span): the full length of the embed call. It contains 1 `ai.embed.doEmbed` spans. It contains the basic embedding span information and the following attributes:

    - `operation.name`: `ai.embed` and the functionId that was set through `telemetry.functionId`
    - `ai.operationId`: `"ai.embed"`
    - `ai.value`: the value that was passed into the `embed` function
    - `ai.embedding`: a JSON-stringified embedding

- `ai.embed.doEmbed` (span): a provider doEmbed call. It contains the basic embedding span information and the following attributes:

    - `operation.name`: `ai.embed.doEmbed` and the functionId that was set through `telemetry.functionId`
    - `ai.operationId`: `"ai.embed.doEmbed"`
    - `ai.values`: the values that were passed into the provider (array)
    - `ai.embeddings`: an array of JSON-stringified embeddings

embedMany function

`embedMany` records 2 types of spans:

- `ai.embedMany` (span): the full length of the embedMany call. It contains 1 or more `ai.embedMany.doEmbed` spans. It contains the basic embedding span information and the following attributes:

    - `operation.name`: `ai.embedMany` and the functionId that was set through `telemetry.functionId`
    - `ai.operationId`: `"ai.embedMany"`
    - `ai.values`: the values that were passed into the `embedMany` function
    - `ai.embeddings`: an array of JSON-stringified embedding

- `ai.embedMany.doEmbed` (span): a provider doEmbed call. It contains the basic embedding span information and the following attributes:
  - `operation.name`: `ai.embedMany.doEmbed` and the functionId that was set through `telemetry.functionId`
  - `ai.operationId`: `"ai.embedMany.doEmbed"`
  - `ai.values`: the values that were sent to the provider
  - `ai.embeddings`: an array of JSON-stringified embeddings for each value

Span Details

Basic LLM span information

Many spans that use LLMs (`ai.generateText`, `ai.generateText.doGenerate`, `ai.streamText`, `ai.streamText.doStream`, `ai.generateObject`, `ai.generateObject.doGenerate`, `ai.streamObject`, `ai.streamObject.doStream`) contain the following attributes:

- `resource.name`: the functionId that was set through `telemetry.functionId`
- `ai.model.id`: the id of the model
- `ai.model.provider`: the provider of the model
- `ai.request.headers.*`: the request headers that were passed in through `headers`
- `ai.response.providerMetadata`: provider specific metadata returned with the generation response
- `ai.settings.maxRetries`: the maximum number of retries that were set
- `ai.telemetry.functionId`: the functionId that was set through `telemetry.functionId`
- `ai.telemetry.metadata.*`: the metadata that was passed in through `telemetry.metadata`
- `ai.usage.completionTokens`: the number of completion tokens that were used
- `ai.usage.promptTokens`: the number of prompt tokens that were used

Call LLM span information

Spans that correspond to individual LLM calls (`ai.generateText.doGenerate`, `ai.streamText.doStream`, `ai.generateObject.doGenerate`, `ai.streamObject.doStream`) contain basic LLM span information and the following attributes:

- `ai.response.model`: the model that was used to generate the response. This can be different from the model that was requested if the provider supports aliases.
- `ai.response.id`: the id of the response. Uses the ID from the provider when available.
- `ai.response.timestamp`: the timestamp of the response. Uses the timestamp from the provider when available.
- Semantic Conventions for GenAI operations
  - `gen_ai.system`: the provider that was used
  - `gen_ai.request.model`: the model that was requested
  - `gen_ai.request.temperature`: the temperature that was set
  - `gen_ai.request.max_tokens`: the maximum number of tokens that were set
  - `gen_ai.request.frequency_penalty`: the frequency penalty that was set
  - `gen_ai.request.presence_penalty`: the presence penalty that was set
  - `gen_ai.request.top_k`: the topK parameter value that was set
  - `gen_ai.request.top_p`: the topP parameter value that was set
  - `gen_ai.request.stop_sequences`: the stop sequences

    - `gen_ai.response.finish_reasons`: the finish reasons that were returned by the provider
    - `gen_ai.response.model`: the model that was used to generate the response. This can be different from the model that was requested if the provider supports aliases.
    - `gen_ai.response.id`: the id of the response. Uses the ID from the provider when available.
    - `gen_ai.usage.input_tokens`: the number of prompt tokens that were used
    - `gen_ai.usage.output_tokens`: the number of completion tokens that were used

Basic embedding span information

Many spans that use embedding models (`ai.embed`, `ai.embed.doEmbed`, `ai.embedMany`, `ai.embedMany.doEmbed`) contain the following attributes:

- `ai.model.id`: the id of the model
- `ai.model.provider`: the provider of the model
- `ai.request.headers.*`: the request headers that were passed in through `headers`
- `ai.settings.maxRetries`: the maximum number of retries that were set
- `ai.telemetry.functionId`: the functionId that was set through `telemetry.functionId`
- `ai.telemetry.metadata.*`: the metadata that was passed in through `telemetry.metadata`
- `ai.usage.tokens`: the number of tokens that were used
- `resource.name`: the functionId that was set through `telemetry.functionId`

Tool call spans

Tool call spans (`ai.toolCall`) contain the following attributes:

- `operation.name`: `"ai.toolCall"`
- `ai.operationId`: `"ai.toolCall"`
- `ai.toolCall.name`: the name of the tool
- `ai.toolCall.id`: the id of the tool call
- `ai.toolCall.args`: the input parameters of the tool call
- `ai.toolCall.result`: the output result of the tool call. Only available if the tool call is successful and the result is serializable.

Generating Structured Data

While text generation can be useful, your use case will likely call for generating structured data.
For example, you might want to extract information from text, classify data, or generate synthetic data.

Many language models are capable of generating structured data, often defined as using "JSON modes" or "tools".
However, you need to manually provide schemas and then validate the generated data as LLMs can produce incorrect or incomplete structured data.

The AI SDK standardises structured object generation across model providers
with the `generateObject`
and `streamObject` functions.
You can use both functions with different output strategies, e.g. `array`, `object`, `enum`, or `no-schema`,
and with different generation modes, e.g. `auto`, `tool`, or `json`.
You can use Zod schemas, Valibot, or JSON schemas to specify the shape of the data that you want,
and the AI model will generate data that conforms to that structure.

You can pass Zod objects directly to the AI SDK functions or use the `zodSchema` helper function.

Generate Object

The `generateObject` generates structured data from a prompt.
The schema is also used to validate the generated data, ensuring type safety and correctness.

```
import { generateObject } from 'ai';

import { z } from 'zod';

const { object } = await generateObject({
  model: 'openai/gpt-4.1',
  schema: z.object({
    recipe: z.object({
      name: z.string(),
      ingredients: z.array(z.object({ name: z.string(), amount: z.string() })),
      steps: z.array(z.string()),
    }),
  }),
  prompt: 'Generate a lasagna recipe.',
});
```

See `generateObject` in action with these examples

Accessing response headers & body

Sometimes you need access to the full response from the model provider,
e.g. to access some provider-specific headers or body content.

You can access the raw response headers and body using the `response` property:

```
import { generateObject } from 'ai';
```

```
const result = await generateObject({
  // ...
});

console.log(JSON.stringify(result.response.headers, null, 2));
console.log(JSON.stringify(result.response.body, null, 2));
```

Stream Object

Given the added complexity of returning structured data, model response time can be unacceptable for your interactive use case.
With the `streamObject` function, you can stream the model's response as it is generated.

```
import { streamObject } from 'ai';

const { partialObjectStream } = streamObject({
  // ...
});

// use partialObjectStream as an async iterable

for await (const partialObject of partialObjectStream) {
  console.log(partialObject);
}
```

You can use `streamObject` to stream generated UIs in combination with React Server Components (see Generative UI) or the `useObject` hook.

See `streamObject` in action with these examples

`onError` callback

`streamObject` immediately starts streaming.
Errors become part of the stream and are not thrown to prevent e.g. servers from crashing.

To log errors, you can provide an `onError` callback that is triggered when an error occurs.

```
import { streamObject } from 'ai';

const result = streamObject({
  // ...
  onError({ error }) {
    console.error(error); // your error logging logic here
  },
});
```

Output Strategy

You can use both functions with different output strategies, e.g. `array`, `object`, `enum`, or `no-schema`.

Object

The default output strategy is `object`, which returns the generated data as an object.
You don't need to specify the output strategy if you want to use the default.

Array

If you want to generate an array of objects, you can set the output strategy to `array`.
When you use the `array` output strategy, the schema specifies the shape of an array element.
With `streamObject`, you can also stream the generated array elements using `elementStream`.

```
import { openai } from '@ai-sdk/openai';
import { streamObject } from 'ai';
import { z } from 'zod';

const { elementStream } = streamObject({
  model: openai('gpt-4.1'),
  output: 'array',
  schema: z.object({
    name: z.string(),
    class: z.string().describe('Character class, e.g. warrior, mage, or thief.'),
    description: z.string(),
  }),
  prompt: 'Generate 3 hero descriptions for a fantasy role playing game.',
});

for await (const hero of elementStream) {
  console.log(hero);
}
```

Enum

If you want to generate a specific enum value, e.g. for classification tasks,
you can set the output strategy to `enum`
and provide a list of possible values in the `enum` parameter.

Enum output is only available with `generateObject`.

```
import { generateObject } from 'ai';

const { object } = await generateObject({
  model: 'openai/gpt-4.1',
  output: 'enum',
  enum: ['action', 'comedy', 'drama', 'horror', 'sci-fi'],
  prompt:
    'Classify the genre of this movie plot: ' +
    '"A group of astronauts travel through a wormhole in search of a ' +
    'new habitable planet for humanity."',
});
```

No Schema

In some cases, you might not want to use a schema,
for example when the data is a dynamic user request.
You can use the `output` setting to set the output format to `no-schema` in those cases
and omit the schema parameter.

```
import { openai } from '@ai-sdk/openai';
import { generateObject } from 'ai';

const { object } = await generateObject({
  model: openai('gpt-4.1'),
  output: 'no-schema',
  prompt: 'Generate a lasagna recipe.',
});
```

Schema Name and Description

You can optionally specify a name and description for the schema. These are used by some providers for additional LLM guidance, e.g. via tool or schema name.

```
import { generateObject } from 'ai';

import { z } from 'zod';

const { object } = await generateObject({
  model: 'openai/gpt-4.1',
  schemaName: 'Recipe',
  schemaDescription: 'A recipe for a dish.',
  schema: z.object({
    name: z.string(),
    ingredients: z.array(z.object({ name: z.string(), amount: z.string() })),
    steps: z.array(z.string()),
  }),
  prompt: 'Generate a lasagna recipe.',
});
```

Error Handling

When `generateObject` cannot generate a valid object, it throws a `AI_NoObjectGeneratedError`.

This error occurs when the AI provider fails to generate a parsable object that conforms to the schema. It can arise due to the following reasons:

- The model failed to generate a response.
- The model generated a response that could not be parsed.
- The model generated a response that could not be validated against the schema.

The error preserves the following information to help you log the issue:

- `text`: The text that was generated by the model. This can be the raw text or the tool call text, depending on the object generation mode.
- `response`: Metadata about the language model response, including response id, timestamp, and model.
- `usage`: Request token usage.
- `cause`: The cause of the error (e.g. a JSON parsing error). You can use this for more detailed error handling.

```
import { generateObject, NoObjectGeneratedError } from 'ai';

try {
```

```
  await generateObject({ model, schema, prompt });
} catch (error) {
  if (NoObjectGeneratedError.isInstance(error)) {
    console.log('NoObjectGeneratedError');
    console.log('Cause:', error.cause);
    console.log('Text:', error.text);
    console.log('Response:', error.response);
    console.log('Usage:', error.usage);
  }
}
```

Repairing Invalid or Malformed JSON

The `repairText` function is experimental and may change in the future.

Sometimes the model will generate invalid or malformed JSON.
You can use the `repairText` function to attempt to repair the JSON.

It receives the error, either a `JSONParseError` or a `TypeValidationError`,
and the text that was generated by the model.
You can then attempt to repair the text and return the repaired text.

```
import { generateObject } from 'ai';

const { object } = await generateObject({
  model,
  schema,
  prompt,
  experimental_repairText: async ({ text, error }) => {
    // example: add a closing brace to the text
    return text + '}';
  },
});
```

Structured outputs with `generateText` and `streamText`

You can generate structured data with `generateText` and `streamText` by using the `experimental_output` setting.

Some models, e.g. those by OpenAI, support structured outputs and tool calling
at the same time. This is only possible with `generateText` and `streamText`.

Structured output generation with `generateText` and `streamText` is
experimental and may change in the future.

`generateText`

```
// experimental_output is a structured object that matches the schema:

const { experimental_output } = await generateText({
  // ...
  experimental_output: Output.object({
```

```
    schema: z.object({
      name: z.string(),
      age: z.number().nullable().describe('Age of the person.'),
      contact: z.object({
        type: z.literal('email'),
        value: z.string(),
      }),
      occupation: z.object({
        type: z.literal('employed'),
        company: z.string(),
        position: z.string(),
      }),
    }),
  }),
  prompt: 'Generate an example person for testing.',
});
```

`streamText`

```
// experimental_partialOutputStream contains generated partial objects:

const { experimental_partialOutputStream } = await streamText({
  // ...
  experimental_output: Output.object({
    schema: z.object({
      name: z.string(),
      age: z.number().nullable().describe('Age of the person.'),
      contact: z.object({
        type: z.literal('email'),
        value: z.string(),
      }),
      occupation: z.object({
        type: z.literal('employed'),
        company: z.string(),
        position: z.string(),
      }),
    }),
  }),
  prompt: 'Generate an example person for testing.',
});
```

More Examples

You can see `generateObject` and `streamObject` in action using various frameworks in the following examples:

`generateObject`

- Learn to generate objects in Node.js
- Learn to generate objects in Next.js with Route Handlers (AI SDK UI)
- Learn to generate objects in Next.js with Server Actions (AI SDK RSC)

`streamObject`

- Learn to stream objects in Node.js
- Learn to stream objects in Next.js with Route Handlers (AI SDK UI)
- Learn to stream objects in Next.js with Server Actions (AI SDK RSC)

Tool Calling

As covered under Foundations, tools are objects that can be called by the model to perform a specific task.
AI SDK Core tools contain three elements:

- `description`: An optional description of the tool that can influence when the tool is picked.
- `inputSchema`: A Zod schema or a JSON schema that defines the input parameters. The schema is consumed by the LLM, and also used to validate the LLM tool calls.
- `execute`: An optional async function that is called with the arguments from the tool call. It produces a value of type `RESULT` (generic type). It is optional because you might want to forward tool calls to the client or to a queue instead of executing them in the same process.

You can use the `tool` helper function to
infer the types of the `execute` parameters.

The `tools` parameter of `generateText` and `streamText` is an object that has the tool names as keys and the tools as values:

```
import { z } from 'zod'
import { generateText, tool } from 'ai'

const result = await generateText({
  model: 'openai/gpt-4o',
  tools: {
    weather: tool({
      description: 'Get the weather in a location',
      inputSchema: z.object({
        location: z.string().describe('The location to get the weather for'),
      }),
      execute: async ({ location }) => ({
        location,
        temperature: 72 + Math.floor(Math.random() * 21) - 10,
      }),
    }),
  },
  prompt: 'What is the weather in San Francisco?',
})
```

When a model uses a tool, it is called a "tool call" and the output of the
tool is called a "tool result".

Tool calling is not restricted to only text generation.
You can also use it to render user interfaces (Generative UI).

Multi-Step Calls (using stopWhen)

With the `stopWhen` setting, you can enable multi-step calls in `generateText` and `streamText`.
When `stopWhen` is set and the model generates a tool call, the AI SDK will trigger a new generation passing in the tool result until there are no further tool calls or the stopping condition is met.

The `stopWhen` conditions are only evaluated when the last step contains tool
results.

By default, when you use `generateText` or `streamText`, it triggers a single generation. This works well for many use cases where you can rely on the model's training data to generate a response. However, when you provide tools, the model now has the choice to either generate a normal text response, or generate a tool call. If the model generates a tool call, it's generation is complete and that step is finished.

You may want the model to generate text after the tool has been executed, either to summarize the tool results in the context of the users query. In many cases, you may also want the model to use multiple tools in a single response. This is where multi-step calls come in.

You can think of multi-step calls in a similar way to a conversation with a human. When you ask a question, if the person does not have the requisite knowledge in their common knowledge (a model's training data), the person may need to look up information (use a tool) before they can provide you with an answer. In the same way, the model may need to call a tool to get the information it needs to answer your question where each generation (tool call or text generation) is a step.

Example

In the following example, there are two steps:

1. **Step 1**
    1. The prompt `'What is the weather in San Francisco?'` is sent to the model.
    2. The model generates a tool call.
    3. The tool call is executed.
2. **Step 2**
    1. The tool result is sent to the model.
    2. The model generates a response considering the tool result.

```
import { z } from 'zod'
import { generateText, tool, stepCountIs } from 'ai'

const { text, steps } = await generateText({
  model: 'openai/gpt-4o',
  tools: {
    weather: tool({
      description: 'Get the weather in a location',
      inputSchema: z.object({
        location: z.string().describe('The location to get the weather for'),
      }),
      execute: async ({ location }) => ({
        location,
        temperature: 72 + Math.floor(Math.random() * 21) - 10,
      }),
    }),
  },
  stopWhen: stepCountIs(5), // stop after 5 steps if tools were called
  prompt: 'What is the weather in San Francisco?',
})
```

You can use `streamText` in a similar way.

Steps

To access intermediate tool calls and results, you can use the `steps` property in the result object or the `streamText` `onFinish` callback.

It contains all the text, tool calls, tool results, and more from each step.

Example: Extract tool results from all steps

```
import { generateText } from 'ai'

const { steps } = await generateText({
  model: openai('gpt-4o'),
  stopWhen: stepCountIs(10),
  // ...
})

// extract all tool calls from the steps:
const allToolCalls = steps.flatMap(step => step.toolCalls)
```

`onStepFinish` callback

When using `generateText` or `streamText`, you can provide an `onStepFinish` callback that is triggered when a step is finished,
i.e. all text deltas, tool calls, and tool results for the step are available.
When you have multiple steps, the callback is triggered for each step.

```
import { generateText } from 'ai'

const result = await generateText({
  // ...
  onStepFinish({ text, toolCalls, toolResults, finishReason, usage }) {
    // your own logic, e.g. for saving the chat history or recording usage
  },
})
```

`prepareStep` callback

The `prepareStep` callback is called before a step is started.

It is called with the following parameters:

- `model`: The model that was passed into `generateText`.
- `stopWhen`: The stopping condition that was passed into `generateText`.
- `stepNumber`: The number of the step that is being executed.
- `steps`: The steps that have been executed so far.
- `messages`: The messages that will be sent to the model for the current step.

You can use it to provide different settings for a step, including modifying the input messages.

```
import { generateText } from 'ai'

const result = await generateText({
  // ...
  prepareStep: async ({ model, stepNumber, steps, messages }) => {
    if (stepNumber === 0) {
      return {
        // use a different model for this step:
        model: modelForThisParticularStep,
        // force a tool choice for this step:
```

```
        toolChoice: { type: 'tool', toolName: 'tool1' },
        // limit the tools that are available for this step:
        activeTools: ['tool1'],
      }
    }

    // when nothing is returned, the default settings are used
  },
})
```

Message Modification for Longer Agentic Loops

In longer agentic loops, you can use the `messages` parameter to modify the input messages for each step. This is particularly useful for prompt compression:

```
prepareStep: async ({ stepNumber, steps, messages }) => {
  // Compress conversation history for longer loops
  if (messages.length > 20) {
    return {
      messages: messages.slice(-10),
    }
  }

  return {};
}
```

Response Messages

Adding the generated assistant and tool messages to your conversation history is a common task, especially if you are using multi-step tool calls.

Both `generateText` and `streamText` have a `response.messages` property that you can use to
add the assistant and tool messages to your conversation history.
It is also available in the `onFinish` callback of `streamText`.

The `response.messages` property contains an array of `ModelMessage` objects that you can add to your conversation history:

```
import { generateText, ModelMessage } from 'ai'

const messages: ModelMessage[] = [
  // ...
]

const { response } = await generateText({
  // ...
  messages,
})

// add the response messages to your conversation history:
messages.push(...response.messages)  // streamText: ...((await response).messages
```

Dynamic Tools

AI SDK Core supports dynamic tools for scenarios where tool schemas are not known at compile time. This is useful for:

- MCP (Model Context Protocol) tools without schemas
- User-defined functions at runtime
- Tools loaded from external sources

Using dynamicTool

The `dynamicTool` helper creates tools with unknown input/output types:

```
import { dynamicTool } from 'ai'
import { z } from 'zod'

const customTool = dynamicTool({
  description: 'Execute a custom function',
  inputSchema: z.object({}),
  execute: async input => {
    // input is typed as 'unknown'
    // You need to validate/cast it at runtime
    const { action, parameters } = input as any;

    // Execute your dynamic logic
    return { result: `Executed ${action}` };
  },
})
```

Type-Safe Handling

When using both static and dynamic tools, use the `dynamic` flag for type narrowing:

```
const result = await generateText({
  model: 'openai/gpt-4o',
  tools: {
    // Static tool with known types
    weather: weatherTool,

    // Dynamic tool
    custom: dynamicTool({
      /* ... */
    }),
  },
  onStepFinish: ({ toolCalls, toolResults }) => {
    // Type-safe iteration
    for (const toolCall of toolCalls) {
      if (toolCall.dynamic) {
        // Dynamic tool: input is 'unknown'
        console.log('Dynamic:', toolCall.toolName, toolCall.input);
        continue;
      }

      // Static tool: full type inference
      switch (toolCall.toolName) {
        case 'weather':
          console.log(toolCall.input.location);  // typed as string
```

```
          break;
      }
    }
  },
});
```

Tool Choice

You can use the `toolChoice` setting to influence when a tool is selected.
It supports the following settings:

- `auto` (default): the model can choose whether and which tools to call.
- `required` : the model must call a tool. It can choose which tool to call.
- `none` : the model must not call tools
- `{ type: 'tool', toolName: string (typed) }` : the model must call the specified tool

```
import { z } from 'zod'
import { generateText, tool } from 'ai'

const result = await generateText({
  model: 'openai/gpt-4o',
  tools: {
    weather: tool({
      description: 'Get the weather in a location',
      inputSchema: z.object({
        location: z.string().describe('The location to get the weather for'),
      }),
      execute: async ({ location }) => ({
        location,
        temperature: 72 + Math.floor(Math.random() * 21) - 10,
      }),
    }),
  },
  toolChoice: 'required',  // force the model to call a tool
  prompt: 'What is the weather in San Francisco?',
})
```

Tool Execution Options

When tools are called, they receive additional options as a second parameter.

Tool Call ID

The ID of the tool call is forwarded to the tool execution.
You can use it e.g. when sending tool-call related information with stream data.

```
import { streamText, tool, createUIMessageStream, createUIMessageStreamResponse }

export async function POST(req: Request) {
  const { messages } = await req.json();

  const stream = createUIMessageStream({
    execute: ({ writer }) => {
      const result = streamText({
```

```
        // ...
        messages,
        tools: {
          myTool: tool({
            // ...
            execute: async (args, { toolCallId }) => {
              // return e.g. custom status for tool call
              writer.write({
                type: 'data-tool-status',
                id: toolCallId,
                data: {
                  name: 'myTool',
                  status: 'in-progress',
                },
              })
              // ...
            },
          }),
        },
      });

      writer.merge(result.toUIMessageStream());
    },
  });

  return createUIMessageStreamResponse({ stream });
}
```

Messages

The messages that were sent to the language model to initiate the response that contained the tool call are forwarded to the tool execution.
You can access them in the second parameter of the `execute` function.
In multi-step calls, the messages contain the text, tool calls, and tool results from all previous steps.

```
import { generateText, tool } from 'ai'

const result = await generateText({
  // ...
  tools: {
    myTool: tool({
      // ...
      execute: async (args, { messages }) => {
        // use the message history in e.g. calls to other language models
        return { ... };
      },
    }),
  },
})
```

Abort Signals

The abort signals from `generateText` and `streamText` are forwarded to the tool execution.
You can access them in the second parameter of the `execute` function and e.g. abort long-running

computations or forward them to fetch calls inside tools.

```
import { z } from 'zod'
import { generateText, tool } from 'ai'

const result = await generateText({
  model: 'openai/gpt-4.1',
  abortSignal: myAbortSignal,  // signal that will be forwarded to tools
  tools: {
    weather: tool({
      description: 'Get the weather in a location',
      inputSchema: z.object({ location: z.string() }),
      execute: async ({ location }, { abortSignal }) => {
        return fetch(
          `https://api.weatherapi.com/v1/current.json?q=${location}`,
          { signal: abortSignal },  // forward the abort signal to fetch
        )
      },
    }),
  },
  prompt: 'What is the weather in San Francisco?',
})
```

Context (experimental)

You can pass in arbitrary context from `generateText` or `streamText` via the `experimental_context` setting.
This context is available in the `experimental_context` tool execution option.

```
const result = await generateText({
  // ...
  tools: {
    someTool: tool({
      // ...
      execute: async (input, { experimental_context: context }) => {
        const typedContext = context as { example: string }; // or use type valid
        // ...
      },
    }),
  },
  experimental_context: { example: '123' },
})
```

Types

Modularizing your code often requires defining types to ensure type safety and reusability.
To enable this, the AI SDK provides several helper types for tools, tool calls, and tool results.

You can use them to strongly type your variables, function parameters, and return types
in parts of the code that are not directly related to `streamText` or `generateText`.

Each tool call is typed with `ToolCall<NAME extends string, ARGS>`, depending
on the tool that has been invoked.
Similarly, the tool results are typed with `ToolResult<NAME extends string, ARGS, RESULT>`.

The tools in `streamText` and `generateText` are defined as a `ToolSet`.
The type inference helpers `TypedToolCall<TOOLS extends ToolSet>`
and `TypedToolResult<TOOLS extends ToolSet>` can be used to
extract the tool call and tool result types from the tools.

```
import { openai } from '@ai-sdk/openai'
import { TypedToolCall, TypedToolResult, generateText, tool } from 'ai'
import { z } from 'zod'

const myToolSet = {
  firstTool: tool({
    description: 'Greets the user',
    inputSchema: z.object({ name: z.string() }),
    execute: async ({ name }) => `Hello, ${name}!`,
  }),
  secondTool: tool({
    description: 'Tells the user their age',
    inputSchema: z.object({ age: z.number() }),
    execute: async ({ age }) => `You are ${age} years old!`,
  }),
}

type MyToolCall = TypedToolCall<typeof myToolSet>
type MyToolResult = TypedToolResult<typeof myToolSet>

async function generateSomething(
  prompt: string,
): Promise<{ text: string; toolCalls: MyToolCall[]; toolResults: MyToolResult[] }
  return generateText({
    model: openai('gpt-4.1'),
    tools: myToolSet,
    prompt,
  })
}
```

Handling Errors

The AI SDK has three tool-call related errors:

- NoSuchToolError: the model tries to call a tool that is not defined in the tools object
- InvalidToolArgumentsError: the model calls a tool with arguments that do not match the tool's input schema
- ToolExecutionError: an error that occurred during tool execution
- ToolCallRepairError: an error that occurred during tool call repair

`generateText`

`generateText` throws errors and can be handled using a `try` / `catch` block:

```
try {
  const result = await generateText({
    //...
  })
} catch (error) {
```

```
  if (NoSuchToolError.isInstance(error)) {
    // handle the no such tool error
  } else if (InvalidToolArgumentsError.isInstance(error)) {
    // handle the invalid tool arguments error
  } else if (ToolExecutionError.isInstance(error)) {
    // handle the tool execution error
  } else {
    // handle other errors
  }
}
```

`streamText`

`streamText` sends the errors as part of the full stream. The error parts contain the error object.

When using `toUIMessageStreamResponse`, you can pass an `onError` function to extract the error message from the error part and forward it as part of the stream response:

```
const result = streamText({
  // ...
})

return result.toUIMessageStreamResponse({
  onError: error => {
    if (NoSuchToolError.isInstance(error)) {
      return 'The model tried to call a unknown tool.'
    } else if (InvalidToolArgumentsError.isInstance(error)) {
      return 'The model called a tool with invalid arguments.'
    } else if (ToolExecutionError.isInstance(error)) {
      return 'An error occurred during tool execution.'
    } else {
      return 'An unknown error occurred.'
    }
  },
})
```

Tool Call Repair

The tool call repair feature is experimental and may change in the future.

Language models sometimes fail to generate valid tool calls,
especially when the input schema is complex or the model is smaller.

You can use the `experimental_repairToolCall` function to attempt to repair the tool call with a custom function.

You can use different strategies to repair the tool call:

- Use a model with structured outputs to generate the arguments.
- Send the messages, system prompt, and tool schema to a stronger model to generate the arguments.
- Provide more specific repair instructions based on which tool was called.

Example: Use a model with structured outputs for repair

```
import { openai } from '@ai-sdk/openai'
import { generateObject, generateText, NoSuchToolError, tool } from 'ai'

const result = await generateText({
  model,
  tools,
  prompt,

  experimental_repairToolCall: async ({ toolCall, tools, inputSchema, error }) =>
    if (NoSuchToolError.isInstance(error)) {
      return null; // do not attempt to fix invalid tool names
    }

    const tool = tools[toolCall.toolName as keyof typeof tools];

    const { object: repairedArgs } = await generateObject({
      model: openai('gpt-4.1'),
      schema: tool.inputSchema,
      prompt: [
        `The model tried to call the tool "${toolCall.toolName}"`,
        ` with the following arguments:`,
        JSON.stringify(toolCall.input),
        `The tool accepts the following schema:`,
        JSON.stringify(inputSchema(toolCall)),
        `Please fix the arguments.`,
      ].join('\n'),
    });

    return { ...toolCall, input: JSON.stringify(repairedArgs) };
  },
});
```

Example: Use the re-ask strategy for repair

```
import { openai } from '@ai-sdk/openai'
import { generateObject, generateText, NoSuchToolError, tool } from 'ai'

const result = await generateText({
  model,
  tools,
  prompt,

  experimental_repairToolCall: async ({ toolCall, tools, error, messages, system
    const result = await generateText({
      model,
      system,
      messages: [
        ...messages,
        {
          role: 'assistant',
          content: [
            {
              type: 'tool-call',
              toolCallId: toolCall.toolCallId,
```

```
              toolName: toolCall.toolName,
              input: toolCall.input,
            },
          ],
        },
        {
          role: 'tool' as const,
          content: [
            {
              type: 'tool-result',
              toolCallId: toolCall.toolCallId,
              toolName: toolCall.toolName,
              output: error.message,
            },
          ],
        },
      ],
      tools,
    })

    const newToolCall = result.toolCalls.find(
      newToolCall => newToolCall.toolName === toolCall.toolName,
    )

    return newToolCall != null
      ? {
          toolCallType: 'function' as const,
          toolCallId: toolCall.toolCallId,
          toolName: toolCall.toolName,
          input: JSON.stringify(newToolCall.input),
        }
      : null
  },
})
```

Active Tools

Language models can only handle a limited number of tools at a time, depending on the model.
To allow for static typing using a large number of tools and limiting the available tools to the model at the same time,
the AI SDK provides the `activeTools` property.

It is an array of tool names that are currently active.
By default, the value is `undefined` and all tools are active.

```
import { openai } from '@ai-sdk/openai'
import { generateText } from 'ai'

const { text } = await generateText({
  model: openai('gpt-4.1'),
  tools: myToolSet,
  activeTools: ['firstTool'],
})
```

Multi-modal Tool Results

Multi-modal tool results are experimental and only supported by Anthropic.

In order to send multi-modal tool results, e.g. screenshots, back to the model,
they need to be converted into a specific format.

AI SDK Core tools have an optional `toModelOutput` function
that converts the tool result into a content part.

Here is an example for converting a screenshot into a content part:

```
const result = await generateText({
  model: anthropic('claude-3-5-sonnet-20241022'),
  tools: {
    computer: anthropic.tools.computer_20241022({
      // ...
      async execute({ action, coordinate, text }) {
        switch (action) {
          case 'screenshot': {
            return {
              type: 'image',
              data: fs.readFileSync('./data/screenshot-editor.png').toString('bas
            }
          }
          default: {
            return `executed ${action}`
          }
        }
      },

      // map to tool result content for LLM consumption:
      toModelOutput(result) {
        return {
          type: 'content',
          value:
            typeof result === 'string'
              ? [{ type: 'text', text: result }]
              : [{ type: 'image', data: result.data, mediaType: 'image/png' }],
        }
      },
    }),
  },
  // ...
})
```

Extracting Tools

Once you start having many tools, you might want to extract them into separate files.
The `tool` helper function is crucial for this, because it ensures correct type inference.

Here is an example of an extracted tool:

`tools/weather-tool.ts`

```
import { tool } from 'ai'
import { z } from 'zod'

// the `tool` helper function ensures correct type inference:
export const weatherTool = tool({
  description: 'Get the weather in a location',
  inputSchema: z.object({
    location: z.string().describe('The location to get the weather for'),
  }),
  execute: async ({ location }) => ({
    location,
    temperature: 72 + Math.floor(Math.random() * 21) - 10,
  }),
})
```

MCP Tools

The MCP tools feature is experimental and may change in the future.

The AI SDK supports connecting to Model Context Protocol (MCP) servers to access their tools. This enables your AI applications to discover and use tools across various services through a standardized interface.

Initializing an MCP Client

Create an MCP client using either:

- `SSE` (Server-Sent Events): Uses HTTP-based real-time communication, better suited for remote servers that need to send data over the network
- `stdio`: Uses standard input and output streams for communication, ideal for local tool servers running on the same machine (like CLI tools or local services)
- Custom transport: Bring your own transport by implementing the `MCPTransport` interface, ideal when implementing transports from MCP's official Typescript SDK (e.g. `StreamableHTTPClientTransport`)

SSE Transport

The SSE can be configured using a simple object with a `type` and `url` property:

```
import { experimental_createMCPClient as createMCPClient } from 'ai'

const mcpClient = await createMCPClient({
  transport: {
    type: 'sse',
    url: 'https://my-server.com/sse',

    // optional: configure HTTP headers, e.g. for authentication
    headers: {
      Authorization: 'Bearer my-api-key',
    },
  },
})
```

Stdio Transport

The Stdio transport requires importing the `StdioMCPTransport` class from the `ai/mcp-stdio` package:

```
import { experimental_createMCPClient as createMCPClient } from 'ai'
import { Experimental_StdioMCPTransport as StdioMCPTransport } from 'ai/mcp-stdio

const mcpClient = await createMCPClient({
  transport: new StdioMCPTransport({
    command: 'node',
    args: ['src/stdio/dist/server.js'],
  }),
})
```

Custom Transport

You can also bring your own transport, as long as it implements the `MCPTransport` interface. Below is an example of using the new `StreamableHTTPClientTransport` from MCP's official Typescript SDK:

```
import { MCPTransport, experimental_createMCPClient as createMCPClient } from 'ai
import { StreamableHTTPClientTransport } from '@modelcontextprotocol/sdk/client/s

const url = new URL('http://localhost:3000/mcp')

const mcpClient = await createMCPClient({
  transport: new StreamableHTTPClientTransport(url, {
    sessionId: 'session_123',
  }),
})
```

The client returned by the `experimental_createMCPClient` function is a lightweight client intended for use in tool conversion. It currently does not support all features of the full MCP client, such as: authorization, session management, resumable streams, and receiving notifications.

Closing the MCP Client

After initialization, you should close the MCP client based on your usage pattern:

- For short-lived usage (e.g., single requests), close the client when the response is finished
- For long-running clients (e.g., command line apps), keep the client open but ensure it's closed when the application terminates

When streaming responses, you can close the client when the LLM response has finished. For example, when using `streamText`, you should use the `onFinish` callback:

```
const mcpClient = await experimental_createMCPClient({
  // ...
})

const tools = await mcpClient.tools()

const result = await streamText({
  model: openai('gpt-4.1'),
  tools,
  prompt: 'What is the weather in Brooklyn, New York?',
```

```
  onFinish: async () => {
    await mcpClient.close()
  },
})
```

When generating responses without streaming, you can use try/finally or cleanup functions in your framework:

```
let mcpClient: MCPClient | undefined

try {
  mcpClient = await experimental_createMCPClient({
    // ...
  })
} finally {
  await mcpClient?.close()
}
```

Using MCP Tools

The client's `tools` method acts as an adapter between MCP tools and AI SDK tools. It supports two approaches for working with tool schemas:

Schema Discovery

The simplest approach where all tools offered by the server are listed, and input parameter types are inferred based the schemas provided by the server:

```
const tools = await mcpClient.tools()
```

Pros:

- Simpler to implement
- Automatically stays in sync with server changes

Cons:

- No TypeScript type safety during development
- No IDE autocompletion for tool parameters
- Errors only surface at runtime
- Loads all tools from the server

Schema Definition

You can also define the tools and their input schemas explicitly in your client code:

```
import { z } from 'zod'

const tools = await mcpClient.tools({
  schemas: {
    'get-data': {
      inputSchema: z.object({
        query: z.string().describe('The data query'),
        format: z.enum(['json', 'text']).optional(),
      }),
    },
```

```
    // For tools with zero arguments, you should use an empty object:
    'tool-with-no-args': {
      inputSchema: z.object({}),
    },
  },
})
```

Pros:

- Control over which tools are loaded
- Full TypeScript type safety
- Better IDE support with autocompletion
- Catch parameter mismatches during development

Cons:

- Need to manually keep schemas in sync with server
- More code to maintain

When you define `schemas`, the client will only pull the explicitly defined tools, even if the server offers additional tools. This can be beneficial for:

- Keeping your application focused on the tools it needs
- Reducing unnecessary tool loading
- Making your tool dependencies explicit

Examples

You can see tools in action using various frameworks in the following examples:

- Learn to use tools in Node.js
- Learn to use tools in Next.js with Route Handlers
- Learn to use MCP tools in Node.js

Previous: Generating Structured Data | Next: Prompt Engineering

Prompt Engineering

Tips

Prompts for Tools

When you create prompts that include tools, getting good results can be tricky as the number and complexity of your tools increases.

Here are a few tips to help you get the best results:

1. Use a model that is strong at tool calling, such as `gpt-4` or `gpt-4.1`. Weaker models will often struggle to call tools effectively and flawlessly.
2. Keep the number of tools low, e.g. to 5 or less.
3. Keep the complexity of the tool parameters low. Complex Zod schemas with many nested and optional elements, unions, etc. can be challenging for the model to work with.
4. Use semantically meaningful names for your tools, parameters, parameter properties, etc. The more information you pass to the model, the better it can understand what you want.
5. Add `.describe("...")` to your Zod schema properties to give the model hints about what a particular property is for.
6. When the output of a tool might be unclear to the model and there are dependencies between tools, use the `description` field of a tool to provide information about the output of the tool execution.
7. You can include example input/outputs of tool calls in your prompt to help the model understand how to use the tools. Keep in mind that the tools work with JSON objects, so the examples should use JSON.

In general, the goal should be to give the model all information it needs in a clear way.

Tool & Structured Data Schemas

The mapping from Zod schemas to LLM inputs (typically JSON schema) is not always straightforward, since the mapping is not one-to-one.

Zod Dates

Zod expects JavaScript Date objects, but models return dates as strings.
You can specify and validate the date format using `z.string().datetime()` or `z.string().date()`,
and then use a Zod transformer to convert the string to a Date object.

```
const result = await generateObject({
  model: openai('gpt-4.1'),
  schema: z.object({
    events: z.array(
      z.object({
        event: z.string(),
        date: z
          .string()
          .date()
          .transform(value => new Date(value)),
      }),
```

```
    ),
  }),
  prompt: 'List 5 important events from the year 2000.',
});
```

Optional Parameters

When working with tools that have optional parameters, you may encounter compatibility issues with certain providers that use strict schema validation.

This is particularly relevant for OpenAI models with structured outputs (strict mode).

For maximum compatibility, optional parameters should use `.nullable()` instead of `.optional()`:

```
// This may fail with strict schema validation
const failingTool = tool({
  description: 'Execute a command',
  inputSchema: z.object({
    command: z.string(),
    workdir: z.string().optional(), // This can cause errors
    timeout: z.string().optional(),
  }),
});

// This works with strict schema validation
const workingTool = tool({
  description: 'Execute a command',
  inputSchema: z.object({
    command: z.string(),
    workdir: z.string().nullable(), // Use nullable instead
    timeout: z.string().nullable(),
  }),
});
```

Temperature Settings

For tool calls and object generation, it's recommended to use `temperature: 0` to ensure deterministic and consistent results:

```
const result = await generateText({
  model: openai('gpt-4o'),
  temperature: 0, // Recommended for tool calls
  tools: {
    myTool: tool({
      description: 'Execute a command',
      inputSchema: z.object({
        command: z.string(),
      }),
    }),
  },
  prompt: 'Execute the ls command',
});
```

Lower temperature values reduce randomness in model outputs, which is particularly important when the model needs to:

- Generate structured data with specific formats
- Make precise tool calls with correct parameters
- Follow strict schemas consistently

Debugging

Inspecting Warnings

Not all providers support all AI SDK features.
Providers either throw exceptions or return warnings when they do not support a feature.
To check if your prompt, tools, and settings are handled correctly by the provider, you can check the call warnings:

```
const result = await generateText({
  model: openai('gpt-4o'),
  prompt: 'Hello, world!',
});

console.log(result.warnings);
```

HTTP Request Bodies

You can inspect the raw HTTP request bodies for models that expose them, e.g. OpenAI.
This allows you to inspect the exact payload that is sent to the model provider in the provider-specific way.

Request bodies are available via the `request.body` property of the response:

```
const result = await generateText({
  model: openai('gpt-4o'),
  prompt: 'Hello, world!',
});

console.log(result.request.body);
```

Settings

Large language models (LLMs) typically provide settings to augment their output.

All AI SDK functions support the following common settings in addition to the model, the prompt, and additional provider-specific settings:

```
const result = await generateText({
  model: 'openai/gpt-4.1',
  maxOutputTokens: 512,
  temperature: 0.3,
  maxRetries: 5,
  prompt: 'Invent a new holiday and describe its traditions.',
});
```

Some providers do not support all common settings. If you use a setting with a provider that does not support it, a warning will be generated. You can check the `warnings` property in the result object to see if any warnings were generated.

`maxOutputTokens`

Maximum number of tokens to generate.

`temperature`

Temperature setting.

The value is passed through to the provider. The range depends on the provider and model. For most providers, `0` means almost deterministic results, and higher values mean more randomness.

It is recommended to set either `temperature` or `topP`, but not both.

In AI SDK 5.0, temperature is no longer set to `0` by default.

`topP`

Nucleus sampling.

The value is passed through to the provider. The range depends on the provider and model. For most providers, nucleus sampling is a number between 0 and 1. E.g. 0.1 would mean that only tokens with the top 10% probability mass are considered.

It is recommended to set either `temperature` or `topP`, but not both.

`topK`

Only sample from the top K options for each subsequent token.

Used to remove "long tail" low probability responses. Recommended for advanced use cases only. You usually only need to use `temperature`.

`presencePenalty`

The presence penalty affects the likelihood of the model to repeat information that is already in the prompt.

The value is passed through to the provider. The range depends on the provider and model. For most providers, `0` means no penalty.

`frequencyPenalty`

The frequency penalty affects the likelihood of the model to repeatedly use the same words or phrases.

The value is passed through to the provider. The range depends on the provider and model. For most providers, `0` means no penalty.

`stopSequences`

The stop sequences to use for stopping the text generation.

If set, the model will stop generating text when one of the stop sequences is generated. Providers may have limits on the number of stop sequences.

`seed`

It is the seed (integer) to use for random sampling. If set and supported by the model, calls will generate deterministic results.

`maxRetries`

Maximum number of retries. Set to 0 to disable retries. Default: `2`.

`abortSignal`

An optional abort signal that can be used to cancel the call.

The abort signal can e.g. be forwarded from a user interface to cancel the call, or to define a timeout.

Example: Timeout

```
const result = await generateText({
  model: openai('gpt-4o'),
  prompt: 'Invent a new holiday and describe its traditions.',
  abortSignal: AbortSignal.timeout(5000), // 5 seconds
});
```

`headers`

Additional HTTP headers to be sent with the request. Only applicable for HTTP-based providers.

You can use the request headers to provide additional information to the provider, depending on what the provider supports. For example, some observability providers support headers such as `Prompt-Id`.

```
import { generateText } from 'ai';
import { openai } from '@ai-sdk/openai';

const result = await generateText({
```

```
  model: openai('gpt-4o'),
  prompt: 'Invent a new holiday and describe its traditions.',
  headers: {
    'Prompt-Id': 'my-prompt-id',
  },
});
```

The `headers` setting is for request-specific headers. You can also set `headers` in the provider configuration. These headers will be sent with every request made by the provider.

Previous: Prompt Engineering | Next: Embeddings

Embeddings

Embeddings are a way to represent words, phrases, or images as vectors in a high-dimensional space. In this space, similar words are close to each other, and the distance between words can be used to measure their similarity.

Embedding a Single Value

The AI SDK provides the `embed` function to embed single values, which is useful for tasks such as finding similar words or phrases or clustering text.
You can use it with embeddings models, e.g. `openai.textEmbeddingModel('text-embedding-3-large')` or `mistral.textEmbeddingModel('mistral-embed')`.

```
import { embed } from 'ai';
import { openai } from '@ai-sdk/openai';

// 'embedding' is a single embedding object (number[])
const { embedding } = await embed({
  model: openai.textEmbeddingModel('text-embedding-3-small'),
  value: 'sunny day at the beach',
});
```

Embedding Many Values

When loading data, e.g. when preparing a data store for retrieval-augmented generation (RAG), it is often useful to embed many values at once (batch embedding).

The AI SDK provides the `embedMany` function for this purpose.
Similar to `embed`, you can use it with embeddings models,
e.g. `openai.textEmbeddingModel('text-embedding-3-large')` or `mistral.textEmbeddingModel('mistral-embed')`.

```
import { openai } from '@ai-sdk/openai';
import { embedMany } from 'ai';

// 'embeddings' is an array of embedding objects (number[][]).
// It is sorted in the same order as the input values.
const { embeddings } = await embedMany({
  model: openai.textEmbeddingModel('text-embedding-3-small'),
  values: [
    'sunny day at the beach',
    'rainy afternoon in the city',
    'snowy night in the mountains',
  ],
});
```

Embedding Similarity

After embedding values, you can calculate the similarity between them using the `cosineSimilarity` function.

This is useful to e.g. find similar words or phrases in a dataset.
You can also rank and filter related items based on their similarity.

```
import { openai } from '@ai-sdk/openai';
import { cosineSimilarity, embedMany } from 'ai';

const { embeddings } = await embedMany({
  model: openai.textEmbeddingModel('text-embedding-3-small'),
  values: ['sunny day at the beach', 'rainy afternoon in the city'],
});

console.log(
  `cosine similarity: ${cosineSimilarity(embeddings[0], embeddings[1])}`
);
```

Token Usage

Many providers charge based on the number of tokens used to generate embeddings.
Both `embed` and `embedMany` provide token usage information in the `usage` property of the result object:

```
import { openai } from '@ai-sdk/openai';
import { embed } from 'ai';

const { embedding, usage } = await embed({
  model: openai.textEmbeddingModel('text-embedding-3-small'),
  value: 'sunny day at the beach',
});

console.log(usage);  // { tokens: 10 }
```

Settings

Provider Options

Embedding model settings can be configured using `providerOptions` for provider-specific parameters:

```
import { openai } from '@ai-sdk/openai';
import { embed } from 'ai';

const { embedding } = await embed({
  model: openai.textEmbeddingModel('text-embedding-3-small'),
  value: 'sunny day at the beach',
  providerOptions: {
    openai: {
      dimensions: 512,  // Reduce embedding dimensions
    },
  },
});
```

Parallel Requests

The `embedMany` function now supports parallel processing with configurable `maxParallelCalls` to optimize performance:

```
import { openai } from '@ai-sdk/openai';
import { embedMany } from 'ai';

const { embeddings, usage } = await embedMany({
  maxParallelCalls: 2,  // Limit parallel requests
  model: openai.textEmbeddingModel('text-embedding-3-small'),
  values: [
    'sunny day at the beach',
    'rainy afternoon in the city',
    'snowy night in the mountains',
  ],
});
```

Retries

Both `embed` and `embedMany` accept an optional `maxRetries` parameter of type `number` that you can use to set the maximum number of retries for the embedding process.
It defaults to `2` retries (3 attempts in total). You can set it to `0` to disable retries.

```
import { openai } from '@ai-sdk/openai';
import { embed } from 'ai';

const { embedding } = await embed({
  model: openai.textEmbeddingModel('text-embedding-3-small'),
  value: 'sunny day at the beach',
  maxRetries: 0,  // Disable retries
});
```

Abort Signals and Timeouts

Both `embed` and `embedMany` accept an optional `abortSignal` parameter of type `AbortSignal`
that you can use to abort the embedding process or set a timeout.

```
import { openai } from '@ai-sdk/openai';
import { embed } from 'ai';

const { embedding } = await embed({
  model: openai.textEmbeddingModel('text-embedding-3-small'),
  value: 'sunny day at the beach',
  abortSignal: AbortSignal.timeout(1000),  // Abort after 1 second
});
```

Custom Headers

Both `embed` and `embedMany` accept an optional `headers` parameter of type `Record<string, string>`
that you can use to add custom headers to the embedding request.

```
import { openai } from '@ai-sdk/openai';
import { embed } from 'ai';
```

```
const { embedding } = await embed({
  model: openai.textEmbeddingModel('text-embedding-3-small'),
  value: 'sunny day at the beach',
  headers: { 'X-Custom-Header': 'custom-value' },
});
```

Response Information

Both `embed` and `embedMany` return response information that includes the raw provider response:

```
import { openai } from '@ai-sdk/openai';
import { embed } from 'ai';

const { embedding, response } = await embed({
  model: openai.textEmbeddingModel('text-embedding-3-small'),
  value: 'sunny day at the beach',
});

console.log(response);  // Raw provider response
```

Embedding Providers & Models

Several providers offer embedding models:

Provider	Model	Embedding Dimensions
OpenAI	`text-embedding-3-large`	3072
OpenAI	`text-embedding-3-small`	1536
OpenAI	`text-embedding-ada-002`	1536
Google Generative AI	`gemini-embedding-001`	3072
Google Generative AI	`text-embedding-004`	768
Mistral	`mistral-embed`	1024
Cohere	`embed-english-v3.0`	1024
Cohere	`embed-multilingual-v3.0`	1024
Cohere	`embed-english-light-v3.0`	384
Cohere	`embed-multilingual-light-v3.0`	384
Cohere	`embed-english-v2.0`	4096
Cohere	`embed-english-light-v2.0`	1024
Cohere	`embed-multilingual-v2.0`	768
Amazon Bedrock	`amazon.titan-embed-text-v1`	1536
Amazon Bedrock	`amazon.titan-embed-text-v2:0`	1024

Previous: Settings | Next: Image Generation

Image Generation

Image generation is an experimental feature.

The AI SDK provides the `generateImage` function to generate images based on a given prompt using an image model.

```
import { experimental_generateImage as generateImage } from 'ai';

import { openai } from '@ai-sdk/openai';

const { image } = await generateImage({
  model: openai.image('dall-e-3'),
  prompt: 'Santa Claus driving a Cadillac',
});
```

You can access the image data using the `base64` or `uint8Array` properties:

```
const base64 = image.base64; // base64 image data

const uint8Array = image.uint8Array; // Uint8Array image data
```

Settings

Size and Aspect Ratio

Depending on the model, you can either specify the size or the aspect ratio.

Size

The size is specified as a string in the format `{width}x{height}`.
Models only support a few sizes, and the supported sizes are different for each model and provider.

```
import { experimental_generateImage as generateImage } from 'ai';

import { openai } from '@ai-sdk/openai';

const { image } = await generateImage({
  model: openai.image('dall-e-3'),
  prompt: 'Santa Claus driving a Cadillac',
  size: '1024x1024',
});
```

Aspect Ratio

The aspect ratio is specified as a string in the format `{width}:{height}`.
Models only support a few aspect ratios, and the supported aspect ratios are different for each model and provider.

```
import { experimental_generateImage as generateImage } from 'ai';

import { vertex } from '@ai-sdk/google-vertex';
```

```
const { image } = await generateImage({
  model: vertex.image('imagen-3.0-generate-002'),
  prompt: 'Santa Claus driving a Cadillac',
  aspectRatio: '16:9',
});
```

Generating Multiple Images

`generateImage` also supports generating multiple images at once:

```
import { experimental_generateImage as generateImage } from 'ai';

import { openai } from '@ai-sdk/openai';

const { images } = await generateImage({
  model: openai.image('dall-e-2'),
  prompt: 'Santa Claus driving a Cadillac',
  n: 4, // number of images to generate
});
```

`generateImage` will automatically call the model as often as needed (in parallel) to generate the requested number of images.

Each image model has an internal limit on how many images it can generate in a single API call. The AI SDK manages this automatically by batching requests appropriately when you request multiple images using the `n` parameter. By default, the SDK uses provider-documented limits (for example, DALL-E 3 can only generate 1 image per call, while DALL-E 2 supports up to 10).

If needed, you can override this behavior using the `maxImagesPerCall` setting when generating your image. This is particularly useful when working with new or custom models where the default batch size might not be optimal:

```
const { images } = await generateImage({
  model: openai.image('dall-e-2'),
  prompt: 'Santa Claus driving a Cadillac',
  maxImagesPerCall: 5, // Override the default batch size
  n: 10, // Will make 2 calls of 5 images each
});
```

Providing a Seed

You can provide a seed to the `generateImage` function to control the output of the image generation process.
If supported by the model, the same seed will always produce the same image.

```
import { experimental_generateImage as generateImage } from 'ai';

import { openai } from '@ai-sdk/openai';

const { image } = await generateImage({
  model: openai.image('dall-e-3'),
  prompt: 'Santa Claus driving a Cadillac',
  seed: 1234567890,
});
```

Provider-specific Settings

Image models often have provider- or even model-specific settings.
You can pass such settings to the `generateImage` function using the `providerOptions` parameter. The options for the provider (`openai` in the example below) become request body properties.

```
import { experimental_generateImage as generateImage } from 'ai';

import { openai } from '@ai-sdk/openai';

const { image } = await generateImage({
  model: openai.image('dall-e-3'),
  prompt: 'Santa Claus driving a Cadillac',
  size: '1024x1024',
  providerOptions: {
    openai: { style: 'vivid', quality: 'hd' },
  },
});
```

Abort Signals and Timeouts

`generateImage` accepts an optional `abortSignal` parameter of type `AbortSignal` that you can use to abort the image generation process or set a timeout.

```
import { openai } from '@ai-sdk/openai';

import { experimental_generateImage as generateImage } from 'ai';

const { image } = await generateImage({
  model: openai.image('dall-e-3'),
  prompt: 'Santa Claus driving a Cadillac',
  abortSignal: AbortSignal.timeout(1000), // Abort after 1 second
});
```

Custom Headers

`generateImage` accepts an optional `headers` parameter of type `Record<string, string>` that you can use to add custom headers to the image generation request.

```
import { openai } from '@ai-sdk/openai';

import { experimental_generateImage as generateImage } from 'ai';

const { image } = await generateImage({
  model: openai.image('dall-e-3'),
  prompt: 'Santa Claus driving a Cadillac',
  headers: { 'X-Custom-Header': 'custom-value' },
});
```

Warnings

If the model returns warnings, e.g. for unsupported parameters, they will be available in the `warnings` property of the response.

```
const { image, warnings } = await generateImage({
  model: openai.image('dall-e-3'),
  prompt: 'Santa Claus driving a Cadillac',
});
```

Additional provider-specific meta data

Some providers expose additional meta data for the result overall or per image.

```
const prompt = 'Santa Claus driving a Cadillac';

const { image, providerMetadata } = await generateImage({
  model: openai.image('dall-e-3'),
  prompt,
});

const revisedPrompt = providerMetadata.openai.images[0]?.revisedPrompt;

console.log({
  prompt,
  revisedPrompt,
});
```

The outer key of the returned `providerMetadata` is the provider name. The inner values are the metadata. An `images` key is always present in the metadata and is an array with the same length as the top level `images` key.

Error Handling

When `generateImage` cannot generate a valid image, it throws a `AI_NoImageGeneratedError`.

This error occurs when the AI provider fails to generate an image. It can arise due to the following reasons:

- The model failed to generate a response
- The model generated a response that could not be parsed

The error preserves the following information to help you log the issue:

- `responses`: Metadata about the image model responses, including timestamp, model, and headers.
- `cause`: The cause of the error. You can use this for more detailed error handling

```
import { generateImage, NoImageGeneratedError } from 'ai';

try {
  await generateImage({ model, prompt });
} catch (error) {
  if (NoImageGeneratedError.isInstance(error)) {
    console.log('NoImageGeneratedError');
    console.log('Cause:', error.cause);
    console.log('Responses:', error.responses);
  }
}
```

Generating Images with Language Models

Some language models such as Google `gemini-2.0-flash-exp` support multi-modal outputs including images.
With such models, you can access the generated images using the `files` property of the response.

```
import { google } from '@ai-sdk/google';

import { generateText } from 'ai';

const result = await generateText({
  model: google('gemini-2.0-flash-exp'),
  providerOptions: {
    google: { responseModalities: ['TEXT', 'IMAGE'] },
  },
  prompt: 'Generate an image of a comic cat',
});

for (const file of result.files) {
  if (file.mediaType.startsWith('image/')) {
    // The file object provides multiple data formats:

    // Access images as base64 string, Uint8Array binary data, or check type

    // - file.base64: string (data URL format)
    // - file.uint8Array: Uint8Array (binary data)
    // - file.mediaType: string (e.g. "image/png")
  }
}
```

Image Models

Provider	Model	Support sizes (`width x height`) or aspect ratios (`width : height`)
xAI Grok	`grok-2-image`	1024x768 (default)
OpenAI	`gpt-image-1`	1024x1024, 1536x1024, 1024x1536
OpenAI	`dall-e-3`	1024x1024, 1792x1024, 1024x1792
OpenAI	`dall-e-2`	256x256, 512x512, 1024x1024
Amazon Bedrock	`amazon.nova-canvas-v1:0`	320-4096 (multiples of 16), 1:4 to 4:1, max 4.2M pixels
Fal	`fal-ai/flux/dev`	1:1, 3:4, 4:3, 9:16, 16:9, 9:21, 21:9
Fal	`fal-ai/flux-lora`	1:1, 3:4, 4:3, 9:16, 16:9, 9:21, 21:9
Fal	`fal-ai/fast-sdxl`	1:1, 3:4, 4:3, 9:16, 16:9, 9:21, 21:9
Fal	`fal-ai/flux-pro/v1.1-ultra`	1:1, 3:4, 4:3, 9:16, 16:9, 9:21, 21:9
Fal	`fal-ai/ideogram/v2`	1:1, 3:4, 4:3, 9:16, 16:9, 9:21, 21:9
Fal	`fal-ai/recraft-v3`	1:1, 3:4, 4:3, 9:16, 16:9, 9:21, 21:9
Fal	`fal-ai/stable-diffusion-3.5-large`	1:1, 3:4, 4:3, 9:16, 16:9, 9:21, 21:9
Fal	`fal-ai/hyper-sdxl`	1:1, 3:4, 4:3, 9:16, 16:9, 9:21, 21:9
DeepInfra	`stabilityai/sd3.5`	1:1, 16:9, 1:9, 3:2, 2:3, 4:5, 5:4, 9:16, 9:21

Provider	Model	Support sizes (`width x height`) or aspect ratios (`width : height`)
DeepInfra	`black-forest-labs/FLUX-1.1-pro`	256-1440 (multiples of 32)
DeepInfra	`black-forest-labs/FLUX-1-schnell`	256-1440 (multiples of 32)
DeepInfra	`black-forest-labs/FLUX-1-dev`	256-1440 (multiples of 32)
DeepInfra	`black-forest-labs/FLUX-pro`	256-1440 (multiples of 32)
DeepInfra	`stabilityai/sd3.5-medium`	1:1, 16:9, 1:9, 3:2, 2:3, 4:5, 5:4, 9:16, 9:21
DeepInfra	`stabilityai/sdxl-turbo`	1:1, 16:9, 1:9, 3:2, 2:3, 4:5, 5:4, 9:16, 9:21
Replicate	`black-forest-labs/flux-schnell`	1:1, 2:3, 3:2, 4:5, 5:4, 16:9, 9:16, 9:21, 21:9
Replicate	`recraft-ai/recraft-v3`	1024x1024, 1365x1024, 1024x1365, 1536x1024, 1024x1536, 1820x1024, 1024x1820, 1024x2048, 2048x1024, 1434x1024, 1024x1434, 1024x1280, 1280x1024, 1024x1707, 1707x1024
Google	`imagen-3.0-generate-002`	1:1, 3:4, 4:3, 9:16, 16:9
Google Vertex	`imagen-3.0-generate-002`	1:1, 3:4, 4:3, 9:16, 16:9
Google Vertex	`imagen-3.0-fast-generate-001`	1:1, 3:4, 4:3, 9:16, 16:9
Fireworks	`accounts/fireworks/models/flux-1-dev-fp8`	1:1, 2:3, 3:2, 4:5, 5:4, 16:9, 9:16, 9:21, 21:9
Fireworks	`accounts/fireworks/models/flux-1-schnell-fp8`	1:1, 2:3, 3:2, 4:5, 5:4, 16:9, 9:16, 9:21, 21:9
Fireworks	`accounts/fireworks/models/playground-v2-5-1024px-aesthetic`	640x1536, 768x1344, 832x1216, 896x1152, 1024x1024, 1152x896, 1216x832, 1344x768, 1536x640
Fireworks	`accounts/fireworks/models/japanese-stable-diffusion-xl`	640x1536, 768x1344, 832x1216, 896x1152, 1024x1024, 1152x896, 1216x832, 1344x768, 1536x640
Fireworks	`accounts/fireworks/models/playground-v2-1024px-aesthetic`	640x1536, 768x1344, 832x1216, 896x1152, 1024x1024, 1152x896, 1216x832, 1344x768, 1536x640
Fireworks	`accounts/fireworks/models/SSD-1B`	640x1536, 768x1344, 832x1216, 896x1152, 1024x1024, 1152x896, 1216x832, 1344x768, 1536x640
Fireworks	`accounts/fireworks/models/stable-diffusion-xl-1024-v1-0`	640x1536, 768x1344, 832x1216, 896x1152, 1024x1024, 1152x896, 1216x832, 1344x768, 1536x640
Luma	`photon-1`	1:1, 3:4, 4:3, 9:16, 16:9, 9:21, 21:9
Luma	`photon-flash-1`	1:1, 3:4, 4:3, 9:16, 16:9, 9:21, 21:9
Together.ai	`stabilityai/stable-diffusion-xl-base-1.0`	512x512, 768x768, 1024x1024

Provider	Model	Support sizes (`width x height`) or aspect ratios (`width : height`)
Together.ai	`black-forest-labs/FLUX.1-dev`	512x512, 768x768, 1024x1024
Together.ai	`black-forest-labs/FLUX.1-dev-lora`	512x512, 768x768, 1024x1024
Together.ai	`black-forest-labs/FLUX.1-schnell`	512x512, 768x768, 1024x1024
Together.ai	`black-forest-labs/FLUX.1-canny`	512x512, 768x768, 1024x1024
Together.ai	`black-forest-labs/FLUX.1-depth`	512x512, 768x768, 1024x1024
Together.ai	`black-forest-labs/FLUX.1-redux`	512x512, 768x768, 1024x1024
Together.ai	`black-forest-labs/FLUX.1.1-pro`	512x512, 768x768, 1024x1024
Together.ai	`black-forest-labs/FLUX.1-pro`	512x512, 768x768, 1024x1024
Together.ai	`black-forest-labs/FLUX.1-schnell-Free`	512x512, 768x768, 1024x1024

Above are a small subset of the image models supported by the AI SDK providers. For more, see the respective provider documentation.

Transcription

Transcription is an experimental feature.

The AI SDK provides the `transcribe` function to transcribe audio using a transcription model.

```
import { experimental_transcribe as transcribe } from 'ai';

import { openai } from '@ai-sdk/openai';

import { readFile } from 'fs/promises';

const transcript = await transcribe({
  model: openai.transcription('whisper-1'),
  audio: await readFile('audio.mp3'),
});
```

The `audio` property can be a `Uint8Array`, `ArrayBuffer`, `Buffer`, `string` (base64 encoded audio data), or a `URL`.

To access the generated transcript:

```
const text = transcript.text; // transcript text e.g. "Hello, world!"

const segments = transcript.segments; // array of segments with start and end tim

const language = transcript.language; // language of the transcript e.g. "en", if

const durationInSeconds = transcript.durationInSeconds; // duration of the transc
```

Settings

Provider-Specific settings

Transcription models often have provider or model-specific settings which you can set using the `providerOptions` parameter.

```
import { experimental_transcribe as transcribe } from 'ai';

import { openai } from '@ai-sdk/openai';

import { readFile } from 'fs/promises';

const transcript = await transcribe({
  model: openai.transcription('whisper-1'),
  audio: await readFile('audio.mp3'),
  providerOptions: {
    openai: {
      timestampGranularities: ['word'],
    },
  },
});
```

Abort Signals and Timeouts

`transcribe` accepts an optional `abortSignal` parameter of type `AbortSignal` that you can use to abort the transcription process or set a timeout.

```
import { openai } from '@ai-sdk/openai';

import { experimental_transcribe as transcribe } from 'ai';

import { readFile } from 'fs/promises';

const transcript = await transcribe({
  model: openai.transcription('whisper-1'),
  audio: await readFile('audio.mp3'),
  abortSignal: AbortSignal.timeout(1000), // Abort after 1 second
});
```

Custom Headers

`transcribe` accepts an optional `headers` parameter of type `Record<string, string>` that you can use to add custom headers to the transcription request.

```
import { openai } from '@ai-sdk/openai';

import { experimental_transcribe as transcribe } from 'ai';

import { readFile } from 'fs/promises';

const transcript = await transcribe({
  model: openai.transcription('whisper-1'),
  audio: await readFile('audio.mp3'),
  headers: { 'X-Custom-Header': 'custom-value' },
});
```

Warnings

Warnings (e.g. unsupported parameters) are available on the `warnings` property.

```
import { openai } from '@ai-sdk/openai';

import { experimental_transcribe as transcribe } from 'ai';

import { readFile } from 'fs/promises';

const transcript = await transcribe({
  model: openai.transcription('whisper-1'),
  audio: await readFile('audio.mp3'),
});

const warnings = transcript.warnings;
```

Error Handling

When `transcribe` cannot generate a valid transcript, it throws a `AI_NoTranscriptGeneratedError`.

This error can arise for any of the following reasons:

- The model failed to generate a response
- The model generated a response that could not be parsed

The error preserves the following information to help you log the issue:

- `responses`: Metadata about the transcription model responses, including timestamp, model, and headers.
- `cause`: The cause of the error. You can use this for more detailed error handling.

```
import {
  experimental_transcribe as transcribe,
  NoTranscriptGeneratedError,
} from 'ai';

import { openai } from '@ai-sdk/openai';

import { readFile } from 'fs/promises';

try {
  await transcribe({
    model: openai.transcription('whisper-1'),
    audio: await readFile('audio.mp3'),
  });
} catch (error) {
  if (NoTranscriptGeneratedError.isInstance(error)) {
    console.log('NoTranscriptGeneratedError');
    console.log('Cause:', error.cause);
    console.log('Responses:', error.responses);
  }
}
```

Transcription Models

Provider	Model
OpenAI	`whisper-1`
OpenAI	`gpt-4o-transcribe`
OpenAI	`gpt-4o-mini-transcribe`
ElevenLabs	`scribe_v1`
ElevenLabs	`scribe_v1_experimental`
Groq	`whisper-large-v3-turbo`
Groq	`distil-whisper-large-v3-en`
Groq	`whisper-large-v3`
Azure OpenAI	`whisper-1`
Azure OpenAI	`gpt-4o-transcribe`
Azure OpenAI	`gpt-4o-mini-transcribe`
Rev.ai	`machine`
Rev.ai	`low_cost`
Rev.ai	`fusion`
Deepgram	`base` (+ variants)

Provider	Model
Deepgram	`enhanced` (+ variants)
Deepgram	`nova` (+ variants)
Deepgram	`nova-2` (+ variants)
Deepgram	`nova-3` (+ variants)
Gladia	`default`
AssemblyAI	`best`
AssemblyAI	`nano`
Fal	`whisper`
Fal	`wizper`

Above are a small subset of the transcription models supported by the AI SDK providers. For more, see the respective provider documentation.

Previous: Image Generation | Next: Speech

Speech

Speech is an experimental feature.

The AI SDK provides the `generateSpeech` function to generate speech from text using a speech model.

```
import { experimental_generateSpeech as generateSpeech } from 'ai';

import { openai } from '@ai-sdk/openai';

import { readFile } from 'fs/promises';

const audio = await generateSpeech({
  model: openai.speech('tts-1'),
  text: 'Hello, world!',
  voice: 'alloy',
});
```

Language Setting

You can specify the language for speech generation (provider support varies):

```
import { experimental_generateSpeech as generateSpeech } from 'ai';

import { lmnt } from '@ai-sdk/lmnt';

const audio = await generateSpeech({
  model: lmnt.speech('aurora'),
  text: 'Hola, mundo!',
  language: 'es', // Spanish
});
```

To access the generated audio:

```
const audio = audio.audioData; // audio data e.g. Uint8Array
```

Settings

Provider-Specific settings

You can set model-specific settings with the `providerOptions` parameter.

```
import { experimental_generateSpeech as generateSpeech } from 'ai';

import { openai } from '@ai-sdk/openai';

import { readFile } from 'fs/promises';

const audio = await generateSpeech({
  model: openai.speech('tts-1'),
  text: 'Hello, world!',
  providerOptions: {
```

```
    openai: {
      // ...
    },
  },
});
```

Abort Signals and Timeouts

`generateSpeech` accepts an optional `abortSignal` parameter of type `AbortSignal` that you can use to abort the speech generation process or set a timeout.

```
import { openai } from '@ai-sdk/openai';

import { experimental_generateSpeech as generateSpeech } from 'ai';

import { readFile } from 'fs/promises';

const audio = await generateSpeech({
  model: openai.speech('tts-1'),
  text: 'Hello, world!',
  abortSignal: AbortSignal.timeout(1000), // Abort after 1 second
});
```

Custom Headers

`generateSpeech` accepts an optional `headers` parameter of type `Record<string, string>` that you can use to add custom headers to the speech generation request.

```
import { openai } from '@ai-sdk/openai';

import { experimental_generateSpeech as generateSpeech } from 'ai';

import { readFile } from 'fs/promises';

const audio = await generateSpeech({
  model: openai.speech('tts-1'),
  text: 'Hello, world!',
  headers: { 'X-Custom-Header': 'custom-value' },
});
```

Warnings

Warnings (e.g. unsupported parameters) are available on the `warnings` property.

```
import { openai } from '@ai-sdk/openai';

import { experimental_generateSpeech as generateSpeech } from 'ai';

import { readFile } from 'fs/promises';

const audio = await generateSpeech({
  model: openai.speech('tts-1'),
  text: 'Hello, world!',
});
```

```
const warnings = audio.warnings;
```

Error Handling

When `generateSpeech` cannot generate a valid audio, it throws an `AI_NoAudioGeneratedError`.

This error can arise for any of the following reasons:

- The model failed to generate a response
- The model generated a response that could not be parsed

The error preserves the following information to help you log the issue:

- `responses`: Metadata about the speech model responses, including timestamp, model, and headers.
- `cause`: The cause of the error. You can use this for more detailed error handling.

```
import {
  experimental_generateSpeech as generateSpeech,
  AI_NoAudioGeneratedError,
} from 'ai';

import { openai } from '@ai-sdk/openai';

import { readFile } from 'fs/promises';

try {
  await generateSpeech({
    model: openai.speech('tts-1'),
    text: 'Hello, world!',
  });
} catch (error) {
  if (AI_NoAudioGeneratedError.isInstance(error)) {
    console.log('AI_NoAudioGeneratedError');
    console.log('Cause:', error.cause);
    console.log('Responses:', error.responses);
  }
}
```

Speech Models

Provider	Model
OpenAI	`tts-1`
OpenAI	`tts-1-hd`
OpenAI	`gpt-4o-mini-tts`
LMNT	`aurora`
LMNT	`blizzard`
Hume	`default`

Above are a small subset of the speech models supported by the AI SDK providers. For more, see the respective provider documentation.

AI SDK UI

AI SDK UI is designed to help you build interactive chat, completion, and assistant applications with ease. It is a **framework-agnostic toolkit**, streamlining the integration of advanced AI functionalities into your applications.

AI SDK UI provides robust abstractions that simplify the complex tasks of managing chat streams and UI updates on the frontend, enabling you to develop dynamic AI-driven interfaces more efficiently. With four main hooks — `useChat`, `useCompletion`, and `useObject` — you can incorporate real-time chat capabilities, text completions, streamed JSON, and interactive assistant features into your app.

- `useChat` offers real-time streaming of chat messages, abstracting state management for inputs, messages, loading, and errors, allowing for seamless integration into any UI design.
- `useCompletion` enables you to handle text completions in your applications, managing the prompt input and automatically updating the UI as new completions are streamed.
- `useObject` is a hook that allows you to consume streamed JSON objects, providing a simple way to handle and display structured data in your application.

These hooks are designed to reduce the complexity and time required to implement AI interactions, letting you focus on creating exceptional user experiences.

UI Framework Support

AI SDK UI supports the following frameworks: React, Svelte, Vue.js, and Angular.
Here is a comparison of the supported functions across these frameworks:

Function	React	Svelte	Vue.js	Angular
useChat	Chat		Chat	
useCompletion	Completion		Completion	
useObject	StructuredObject		StructuredObject	

Contributions are welcome to implement missing features for non-React frameworks.

Framework Examples

Explore these example implementations for different frameworks:

- **Next.js**
- **Nuxt**
- **SvelteKit**
- **Angular**

API Reference

Please check out the AI SDK UI API Reference for more details on each function.

Chatbot

The `useChat` hook makes it effortless to create a conversational user interface for your chatbot application. It enables the streaming of chat messages from your AI provider, manages the chat state, and updates the UI automatically as new messages arrive.

To summarize, the `useChat` hook provides the following features:

- **Message Streaming**: All the messages from the AI provider are streamed to the chat UI in real-time.
- **Managed States**: The hook manages the states for input, messages, status, error and more for you.
- **Seamless Integration**: Easily integrate your chat AI into any design or layout with minimal effort.

In this guide, you will learn how to use the `useChat` hook to create a chatbot application with real-time message streaming.
Check out our chatbot with tools guide to learn how to use tools in your chatbot.
Let's start with the following example first.

Example

```
'use client';

import { useChat } from '@ai-sdk/react';
import { DefaultChatTransport } from 'ai';
import { useState } from 'react';

export default function Page() {
  const { messages, sendMessage, status } = useChat({
    transport: new DefaultChatTransport({
      api: '/api/chat',
    }),
  });
  const [input, setInput] = useState('');

  return (
    <>
      {messages.map((message) => (
        <div key={message.id}>
          {message.role === 'user' ? 'User: ' : 'AI: '}
          {message.parts.map((part, index) =>
            part.type === 'text' ? <span key={index}>{part.text}</span> : null
          )}
        </div>
      ))}

      <form
        onSubmit={(e) => {
          e.preventDefault();
          if (input.trim()) {
            sendMessage({ text: input });
            setInput('');
          }
```

```
        }}
      >
        <input
          value={input}
          onChange={(e) => setInput(e.target.value)}
          disabled={status !== 'ready'}
          placeholder="Say something..."
        />
        <button type="submit" disabled={status !== 'ready'}>
          Submit
        </button>
      </form>
    </>
  );
}

// app/api/chat/route.ts

import { openai } from '@ai-sdk/openai';
import { convertToModelMessages, streamText, UIMessage } from 'ai';

// Allow streaming responses up to 30 seconds
export const maxDuration = 30;

export async function POST(req: Request) {
  const { messages }: { messages: UIMessage[] } = await req.json();

  const result = streamText({
    model: openai('gpt-4.1'),
    system: 'You are a helpful assistant.',
    messages: convertToModelMessages(messages),
  });

  return result.toUIMessageStreamResponse();
}
```

The UI messages have a new `parts` property that contains the message parts.
We recommend rendering the messages using the `parts` property instead of the `content` property.
The parts property supports different message types, including text, tool invocation, and tool result, and allows for more flexible and complex chat UIs.

In the `Page` component, the `useChat` hook will request to your AI provider endpoint whenever the user sends a message using `sendMessage`.
The messages are then streamed back in real-time and displayed in the chat UI.

This enables a seamless chat experience where the user can see the AI response as soon as it is available,
without having to wait for the entire response to be received.

Customized UI

`useChat` also provides ways to manage the chat message states via code, show status, and update messages without being triggered by user interactions.

Status

The `useChat` hook returns a `status`. It has the following possible values:

- `submitted`: The message has been sent to the API and we're awaiting the start of the response stream.
- `streaming`: The response is actively streaming in from the API, receiving chunks of data.
- `ready`: The full response has been received and processed; a new user message can be submitted.
- `error`: An error occurred during the API request, preventing successful completion.

You can use `status` for e.g. the following purposes:

- To show a loading spinner while the chatbot is processing the user's message.
- To show a "Stop" button to abort the current message.
- To disable the submit button.

```
'use client';

import { useChat } from '@ai-sdk/react';
import { DefaultChatTransport } from 'ai';
import { useState } from 'react';

export default function Page() {
  const { messages, sendMessage, status, stop } = useChat({
    transport: new DefaultChatTransport({
      api: '/api/chat',
    }),
  });
  const [input, setInput] = useState('');

  return (
    <>
      {messages.map((message) => (
        <div key={message.id}>
          {message.role === 'user' ? 'User: ' : 'AI: '}
          {message.parts.map((part, index) =>
            part.type === 'text' ? <span key={index}>{part.text}</span> : null
          )}
        </div>
      ))}

      {(status === 'submitted' || status === 'streaming') && (
        <div>
          {status === 'submitted' && <Spinner />}
          <button type="button" onClick={() => stop()}>
            Stop
          </button>
        </div>
      )}

      <form
        onSubmit={(e) => {
          e.preventDefault();
          if (input.trim()) {
```

```
            sendMessage({ text: input });
            setInput('');
          }
        }}
      >
        <input
          value={input}
          onChange={(e) => setInput(e.target.value)}
          disabled={status !== 'ready'}
          placeholder="Say something..."
        />
        <button type="submit" disabled={status !== 'ready'}>
          Submit
        </button>
      </form>
    </>
  );
}
```

Error State

Similarly, the `error` state reflects the error object thrown during the fetch request.
It can be used to display an error message, disable the submit button, or show a retry button:

> We recommend showing a generic error message to the user, such as "Something went wrong." This is a good practice to avoid leaking information from the server.

```
'use client';

import { useChat } from '@ai-sdk/react';
import { DefaultChatTransport } from 'ai';
import { useState } from 'react';

export default function Chat() {
  const { messages, sendMessage, error, reload } = useChat({
    transport: new DefaultChatTransport({
      api: '/api/chat',
    }),
  });
  const [input, setInput] = useState('');

  return (
    <div>
      {messages.map((m) => (
        <div key={m.id}>
          {m.role}: {' '}
          {m.parts.map((part, index) =>
            part.type === 'text' ? <span key={index}>{part.text}</span> : null
          )}
        </div>
      ))}

      {error && (
        <>
          <div>An error occurred.</div>
```

```
          <button type="button" onClick={reload}>
            Retry
          </button>
        </>
      )}

      <form
        onSubmit={(e) => {
          e.preventDefault();
          if (input.trim()) {
            sendMessage({ text: input });
            setInput('');
          }
        }}
      >
        <input
          value={input}
          onChange={(e) => setInput(e.target.value)}
          disabled={error != null}
        />
      </form>
    </div>
  );
}
```

Please also see the error handling guide for more information.

Modify messages

Sometimes, you may want to directly modify some existing messages. For example, a delete button can be added to each message to allow users to remove them from the chat history.

The `setMessages` function can help you achieve these tasks:

```
const { messages, setMessages } = useChat();

const handleDelete = (id) => {
  setMessages(messages.filter(message => message.id !== id));
};

return (
  <>
    {messages.map((message) => (
      <div key={message.id}>
        {message.role === 'user' ? 'User: ' : 'AI: '}
        {message.parts.map((part, index) =>
          part.type === 'text' ? (
            <span key={index}>{part.text}</span>
          ) : null
        )}
        <button onClick={() => handleDelete(message.id)}>Delete</button>
      </div>
    ))}
    ...
```

```
  </>
);
```

You can think of `messages` and `setMessages` as a pair of `state` and `setState` in React.

Cancellation and regeneration

It's also a common use case to abort the response message while it's still streaming back from the AI provider. You can do this by calling the `stop` function returned by the `useChat` hook.

```
const { stop, status } = useChat();

return (
  <>
    <button onClick={stop} disabled={!(status === 'streaming' || status === 'subm
      Stop
    </button>
    ...
  </>
);
```

When the user clicks the "Stop" button, the fetch request will be aborted. This avoids consuming unnecessary resources and improves the UX of your chatbot application.

Similarly, you can also request the AI provider to reprocess the last message by calling the `regenerate` function returned by the `useChat` hook:

```
const { regenerate, status } = useChat();

return (
  <>
    <button onClick={regenerate} disabled={!(status === 'ready' || status === 'er
      Regenerate
    </button>
    ...
  </>
);
```

When the user clicks the "Regenerate" button, the AI provider will regenerate the last message and replace the current one correspondingly.

Throttling UI Updates

> This feature is currently only available for React.

By default, the `useChat` hook will trigger a render every time a new chunk is received.
You can throttle the UI updates with the `experimental_throttle` option.

```
const { messages, ... } = useChat({
  // Throttle the messages and data updates to 50ms:
  experimental_throttle: 50,
});
```

Event Callbacks

`useChat` provides optional event callbacks that you can use to handle different stages of the chatbot lifecycle:

- `onFinish`: Called when the assistant message is completed
- `onError`: Called when an error occurs during the fetch request.
- `onData`: Called whenever a data part is received.

These callbacks can be used to trigger additional actions, such as logging, analytics, or custom UI updates.

```
import { UIMessage } from 'ai';

const {
  /* ... */
} = useChat({
  onFinish: (message, { usage, finishReason }) => {
    console.log('Finished streaming message:', message);
    console.log('Token usage:', usage);
    console.log('Finish reason:', finishReason);
  },
  onError: (error) => {
    console.error('An error occurred:', error);
  },
  onData: (data) => {
    console.log('Received data part from server:', data);
  },
});
```

It's worth noting that you can abort the processing by throwing an error in the `onData` callback. This will trigger the `onError` callback and stop the message from being appended to the chat UI. This can be useful for handling unexpected responses from the AI provider.

Request Configuration

Custom headers, body, and credentials

By default, the `useChat` hook sends a HTTP POST request to the `/api/chat` endpoint with the message list as the request body. You can customize the request in two ways:

Hook-Level Configuration (Applied to all requests)

You can configure transport-level options that will be applied to all requests made by the hook:

```
import { useChat } from '@ai-sdk/react';
import { DefaultChatTransport } from 'ai';

const { messages, sendMessage } = useChat({
  transport: new DefaultChatTransport({
    api: '/api/custom-chat',
    headers: {
      Authorization: 'your_token',
    },
    body: {
      user_id: '123',
    },
```

```
    credentials: 'same-origin',
  }),
});
```

Dynamic Hook-Level Configuration

You can also provide functions that return configuration values. This is useful for authentication tokens that need to be refreshed, or for configuration that depends on runtime conditions:

```
import { useChat } from '@ai-sdk/react';
import { DefaultChatTransport } from 'ai';

const { messages, sendMessage } = useChat({
  transport: new DefaultChatTransport({
    api: '/api/custom-chat',
    headers: () => ({
      Authorization: `Bearer ${getAuthToken()}`,
      'X-User-ID': getCurrentUserId(),
    }),
    body: () => ({
      sessionId: getCurrentSessionId(),
      preferences: getUserPreferences(),
    }),
    credentials: () => 'include',
  }),
});
```

For component state that changes over time, use `useRef` to store the current value and reference `ref.current` in your configuration function, or prefer request-level options (see next section) for better reliability.

Request-Level Configuration (Recommended)

Recommended: Use request-level options for better flexibility and control. Request-level options take precedence over hook-level options and allow you to customize each request individually.

```
// Pass options as the second parameter to sendMessage
sendMessage(
  { text: input },
  {
    headers: {
      Authorization: 'Bearer token123',
      'X-Custom-Header': 'custom-value',
    },
    body: {
      temperature: 0.7,
      max_tokens: 100,
      user_id: '123',
    },
    metadata: {
      userId: 'user123',
      sessionId: 'session456',
    },
```

```
  }
);
```

The request-level options are merged with hook-level options, with request-level options taking precedence. On your server side, you can handle the request with this additional information.

Setting custom body fields per request

You can configure custom `body` fields on a per-request basis using the second parameter of the `sendMessage` function.
This is useful if you want to pass in additional information to your backend that is not part of the message list.

```
'use client';

import { useChat } from '@ai-sdk/react';
import { useState } from 'react';

export default function Chat() {
  const { messages, sendMessage } = useChat();
  const [input, setInput] = useState('');

  return (
    <div>
      {messages.map((m) => (
        <div key={m.id}>
          {m.role}: {' '}
          {m.parts.map((part, index) =>
            part.type === 'text' ? <span key={index}>{part.text}</span> : null
          )}
        </div>
      ))}

      <form
        onSubmit={(event) => {
          event.preventDefault();
          if (input.trim()) {
            sendMessage(
              { text: input },
              {
                body: {
                  customKey: 'customValue',
                },
              }
            );
            setInput('');
          }
        }}
      >
        <input
          value={input}
          onChange={(e) => setInput(e.target.value)}
        />
      </form>
    </div>
```

```
  );
}
```

You can retrieve these custom fields on your server side by destructuring the request body:

```
export async function POST(req: Request) {
  // Extract additional information ("customKey") from the body of the request:
  const { messages, customKey }: { messages: UIMessage[]; customKey: string } = a
  //...
}
```

Message Metadata

You can attach custom metadata to messages for tracking information like timestamps, model details, and token usage.

```
// Server: Send metadata about the message
return result.toUIMessageStreamResponse({
  messageMetadata: ({ part }) => {
    if (part.type === 'start') {
      return {
        createdAt: Date.now(),
        model: 'gpt-4o',
      };
    }
    if (part.type === 'finish') {
      return {
        totalTokens: part.totalUsage.totalTokens,
      };
    }
  },
});
```

```
// Client: Access metadata via message.metadata
{
  messages.map((message) => (
    <div key={message.id}>
      {message.role}:{' '}
      {message.metadata?.createdAt && (
        new Date(message.metadata.createdAt).toLocaleTimeString()
      )}
      {/* Render message content */}
      {message.parts.map((part, index) =>
        part.type === 'text' ? <span key={index}>{part.text}</span> : null
      )}
      {/* Show token count if available */}
      {message.metadata?.totalTokens && (
        <span>{message.metadata.totalTokens} tokens</span>
      )}
    </div>
  ));
}
```

For complete examples with type safety and advanced use cases, see the Message Metadata documentation.

Transport Configuration

You can configure custom transport behavior using the `transport` option to customize how messages are sent to your API:

```
import { useChat } from '@ai-sdk/react';
import { DefaultChatTransport } from 'ai';

export default function Chat() {
  const { messages, sendMessage } = useChat({
    id: 'my-chat',
    transport: new DefaultChatTransport({
      prepareSendMessagesRequest: ({ id, messages }) => {
        return {
          body: {
            id,
            message: messages[messages.length - 1],
          },
        };
      },
    }),
  });

  // ... rest of your component
}
```

The corresponding API route receives the custom request format:

```
export async function POST(req: Request) {
  const { id, message } = await req.json();

  // Load existing messages and add the new one
  const messages = await loadMessages(id);
  messages.push(message);

  const result = streamText({
    model: openai('gpt-4.1'),
    messages: convertToModelMessages(messages),
  });

  return result.toUIMessageStreamResponse();
}
```

Advanced: Trigger-based routing

For more complex scenarios like message regeneration, you can use trigger-based routing:

```
import { useChat } from '@ai-sdk/react';
import { DefaultChatTransport } from 'ai';

export default function Chat() {
  const { messages, sendMessage, regenerate } = useChat({
    id: 'my-chat',
    transport: new DefaultChatTransport({
      prepareSendMessagesRequest: ({ id, messages, trigger, messageId }) => {
```

```
        if (trigger === 'submit-user-message') {
          return {
            body: {
              trigger: 'submit-user-message',
              id,
              message: messages[messages.length - 1],
              messageId,
            },
          };
        } else if (trigger === 'regenerate-assistant-message') {
          return {
            body: {
              trigger: 'regenerate-assistant-message',
              id,
              messageId,
            },
          };
        }
        throw new Error(`Unsupported trigger: ${trigger}`);
      },
    }),
  });

  // ... rest of your component
}
```

The corresponding API route would handle different triggers:

```
export async function POST(req: Request) {
  const { trigger, id, message, messageId } = await req.json();

  const chat = await readChat(id);

  let messages = chat.messages;

  if (trigger === 'submit-user-message') {
    // Handle new user message
    messages = [...messages, message];
  } else if (trigger === 'regenerate-assistant-message') {
    // Handle message regeneration - remove messages after messageId
    const messageIndex = messages.findIndex(m => m.id === messageId);
    if (messageIndex !== -1) {
      messages = messages.slice(0, messageIndex);
    }
  }

  const result = streamText({
    model: openai('gpt-4.1'),
    messages: convertToModelMessages(messages),
  });

  return result.toUIMessageStreamResponse();
}
```

To learn more about building custom transports, refer to the Transport API documentation.

Controlling the response stream

With `streamText`, you can control how error messages and usage information are sent back to the client.

Error Messages

By default, the error message is masked for security reasons.
The default error message is "An error occurred."
You can forward error messages or send your own error message by providing a `getErrorMessage` function:

```
import { openai } from '@ai-sdk/openai';
import { convertToModelMessages, streamText, UIMessage } from 'ai';

export async function POST(req: Request) {
  const { messages }: { messages: UIMessage[] } = await req.json();

  const result = streamText({
    model: openai('gpt-4.1'),
    messages: convertToModelMessages(messages),
  });

  return result.toUIMessageStreamResponse({
    onError: (error) => {
      if (error == null) {
        return 'unknown error';
      }
      if (typeof error === 'string') {
        return error;
      }
      if (error instanceof Error) {
        return error.message;
      }
      return JSON.stringify(error);
    },
  });
}
```

Usage Information

By default, the usage information is sent back to the client. You can disable it by setting the `sendUsage` option to `false`:

```
import { openai } from '@ai-sdk/openai';
import { convertToModelMessages, streamText, UIMessage } from 'ai';

export async function POST(req: Request) {
  const { messages }: { messages: UIMessage[] } = await req.json();

  const result = streamText({
    model: openai('gpt-4.1'),
    messages: convertToModelMessages(messages),
  });
```

```
  return result.toUIMessageStreamResponse({
    sendUsage: false,
  });
}
```

Text Streams

`useChat` can handle plain text streams by setting the `streamProtocol` option to `text`:

```
'use client';

import { useChat } from '@ai-sdk/react';
import { TextStreamChatTransport } from 'ai';

export default function Chat() {
  const { messages } = useChat({
    transport: new TextStreamChatTransport({
      api: '/api/chat',
    }),
  });

  return <>...</>;
}
```

This configuration also works with other backend servers that stream plain text.
Check out the stream protocol guide for more information.

> When using `TextStreamChatTransport`, tool calls, usage information and finish reasons are not available.

Reasoning

Some models such as DeepSeek `deepseek-reasoner` and Anthropic `claude-3-7-sonnet-20250219` support reasoning tokens.
These tokens are typically sent before the message content.
You can forward them to the client with the `sendReasoning` option:

```
import { deepseek } from '@ai-sdk/deepseek';
import { convertToModelMessages, streamText, UIMessage } from 'ai';

export async function POST(req: Request) {
  const { messages }: { messages: UIMessage[] } = await req.json();

  const result = streamText({
    model: deepseek('deepseek-reasoner'),
    messages: convertToModelMessages(messages),
  });

  return result.toUIMessageStreamResponse({
    sendReasoning: true,
  });
}
```

On the client side, you can access the reasoning parts of the message object.
Reasoning parts have a `text` property that contains the reasoning content.

```
messages.map((message) => (
  <div key={message.id}>
    {message.role === 'user' ? 'User: ' : 'AI: '}
    {message.parts.map((part, index) => {
      // text parts:
      if (part.type === 'text') {
        return <div key={index}>{part.text}</div>;
      }

      // reasoning parts:
      if (part.type === 'reasoning') {
        return <pre key={index}>{part.text}</pre>;
      }
    })}
  </div>
));
```

Sources

Some providers such as Perplexity and
Google Generative AI include sources in the response.

Currently sources are limited to web pages that ground the response.
You can forward them to the client with the `sendSources` option:

```
import { perplexity } from '@ai-sdk/perplexity';
import { convertToModelMessages, streamText, UIMessage } from 'ai';

export async function POST(req: Request) {
  const { messages }: { messages: UIMessage[] } = await req.json();

  const result = streamText({
    model: perplexity('sonar-pro'),
    messages: convertToModelMessages(messages),
  });

  return result.toUIMessageStreamResponse({
    sendSources: true,
  });
}
```

On the client side, you can access source parts of the message object.
There are two types of sources: `source-url` for web pages and `source-document` for documents.
Here is an example that renders both types of sources:

```
messages.map((message) => (
  <div key={message.id}>
    {message.role === 'user' ? 'User: ' : 'AI: '}

    {/* Render URL sources */}
    {message.parts
```

```
      .filter((part) => part.type === 'source-url')
      .map((part) => (
        <span key={`source-${part.id}`}>
          [
          <a href={part.url} target="_blank">
            {part.title ?? new URL(part.url).hostname}
          </a>
          ]
        </span>
      ))}

    {/* Render document sources */}
    {message.parts
      .filter((part) => part.type === 'source-document')
      .map((part) => (
        <span key={`source-${part.id}`}>
          [<span>{part.title ?? `Document ${part.id}`}</span>]
        </span>
      ))}
  </div>
));
```

Image Generation

Some models such as Google `gemini-2.0-flash-exp` support image generation.
When images are generated, they are exposed as files to the client.
On the client side, you can access file parts of the message object and render them as images.

```
messages.map((message) => (
  <div key={message.id}>
    {message.role === 'user' ? 'User: ' : 'AI: '}
    {message.parts.map((part, index) => {
      if (part.type === 'text') {
        return <div key={index}>{part.text}</div>;
      } else if (part.type === 'file' && part.mediaType.startsWith('image/')) {
        return <img key={index} src={part.url} alt="Generated image" />;
      }
      return null;
    })}
  </div>
));
```

Attachments

The `useChat` hook supports sending file attachments along with a message as well as rendering them on the client. This can be useful for building applications that involve sending images, files, or other media content to the AI provider.

There are two ways to send files with a message: using a `FileList` object from file inputs or using an array of file objects.

FileList

By using `FileList`, you can send multiple files as attachments along with a message using the file input element. The `useChat` hook will automatically convert them into data URLs and send them to the AI provider.

> Currently, only `image/*` and `text/*` content types get automatically converted into multi-modal content parts. You will need to handle other content types manually.

```
'use client';

import { useChat } from '@ai-sdk/react';
import { useRef, useState } from 'react';

export default function Page() {
  const { messages, sendMessage, status } = useChat();
  const [input, setInput] = useState('');
  const [files, setFiles] = useState<FileList | undefined>();
  const fileInputRef = useRef<HTMLInputElement>(null);

  return (
    <div>
      <div>
        {messages.map((message) => (
          <div key={message.id}>
            <div>{`${message.role}: `}</div>
            <div>
              {message.parts.map((part, index) => {
                if (part.type === 'text') {
                  return <span key={index}>{part.text}</span>;
                }
                if (part.type === 'file' && part.mediaType?.startsWith('image/'))
                  return <img key={index} src={part.url} alt={part.filename} />;
                }
                return null;
              })}
            </div>
          </div>
        ))}
      </div>

      <form
        onSubmit={(event) => {
          event.preventDefault();
          if (input.trim()) {
            sendMessage({ text: input, files });
            setInput('');
            setFiles(undefined);
            if (fileInputRef.current) {
              fileInputRef.current.value = '';
            }
          }
        }}
      >
        <input
          type="file"
```

```
          onChange={(event) => {
            if (event.target.files) {
              setFiles(event.target.files);
            }
          }}
          multiple
          ref={fileInputRef}
        />
        <input
          value={input}
          placeholder="Send message..."
          onChange={(e) => setInput(e.target.value)}
          disabled={status !== 'ready'}
        />
      </form>
    </div>
  );
}
```

File Objects

You can also send files as objects along with a message. This can be useful for sending pre-uploaded files or data URLs.

```
'use client';

import { useChat } from '@ai-sdk/react';
import { useState } from 'react';
import { FileUIPart } from 'ai';

export default function Page() {
  const { messages, sendMessage, status } = useChat();
  const [input, setInput] = useState('');
  const [files] = useState<FileUIPart[]>([
    {
      type: 'file',
      filename: 'earth.png',
      mediaType: 'image/png',
      url: 'https://example.com/earth.png',
    },
    {
      type: 'file',
      filename: 'moon.png',
      mediaType: 'image/png',
      url: '...',
    },
  ]);

  return (
    <div>
      <div>
        {messages.map((message) => (
          <div key={message.id}>
            <div>{`${message.role}: `}</div>
```

```
            <div>
              {message.parts.map((part, index) => {
                if (part.type === 'text') {
                  return <span key={index}>{part.text}</span>;
                }
                if (part.type === 'file' && part.mediaType?.startsWith('image/'))
                  return <img key={index} src={part.url} alt={part.filename} />;
                }
                return null;
              })}
            </div>
          </div>
        ))}
      </div>

      <form
        onSubmit={(event) => {
          event.preventDefault();
          if (input.trim()) {
            sendMessage({ text: input, files });
            setInput('');
          }
        }}
      >
        <input
          value={input}
          placeholder="Send message..."
          onChange={(e) => setInput(e.target.value)}
          disabled={status !== 'ready'}
        />
      </form>
    </div>
  );
}
```

Type Inference for Tools

When working with tools in TypeScript, AI SDK UI provides type inference helpers to ensure type safety for your tool inputs and outputs.

InferUITool

The `InferUITool` type helper infers the input and output types of a single tool for use in UI messages:

```
import { InferUITool } from 'ai';
import { z } from 'zod';

const weatherTool = {
  description: 'Get the current weather',
  parameters: z.object({
    location: z.string().describe('The city and state'),
  }),
  execute: async ({ location }) => {
```

```
    return `The weather in ${location} is sunny.`;
  },
};

// Infer the types from the tool
type WeatherUITool = InferUITool<typeof weatherTool>;

// This creates a type with:

// {
//   input: { location: string };
//   output: string;
// }
```

InferUITools

The `InferUITools` type helper infers the input and output types of a `ToolSet`:

```
import { InferUITools, ToolSet } from 'ai';
import { z } from 'zod';

const tools: ToolSet = {
  weather: {
    description: 'Get the current weather',
    parameters: z.object({
      location: z.string().describe('The city and state'),
    }),
    execute: async ({ location }) => {
      return `The weather in ${location} is sunny.`;
    },
  },
  calculator: {
    description: 'Perform basic arithmetic',
    parameters: z.object({
      operation: z.enum(['add', 'subtract', 'multiply', 'divide']),
      a: z.number(),
      b: z.number(),
    }),
    execute: async ({ operation, a, b }) => {
      switch (operation) {
        case 'add':
          return a + b;
        case 'subtract':
          return a - b;
        case 'multiply':
          return a * b;
        case 'divide':
          return a / b;
      }
    },
  },
};

// Infer the types from the tool set
```

```
type MyUITools = InferUITools<typeof tools>;

// This creates a type with:

// {
//   weather: { input: { location: string }; output: string };
//   calculator: { input: { operation: 'add' | 'subtract' | 'multiply' | 'divide'
// }
```

Using Inferred Types

You can use these inferred types to create a custom `UIMessage` type and pass it to various AI SDK UI functions:

```
import { InferUITools, UIMessage, UIDataTypes } from 'ai';

type MyUITools = InferUITools<typeof tools>;

type MyUIMessage = UIMessage<never, UIDataTypes, MyUITools>;
```

Pass the custom type to `useChat` or `createUIMessageStream`:

```
import { useChat } from '@ai-sdk/react';
import { createUIMessageStream } from 'ai';
import { MyUIMessage } from './types';

// With useChat
const { messages } = useChat<MyUIMessage>();

// With createUIMessageStream
const stream = createUIMessageStream<MyUIMessage>(/* ... */);
```

This provides full type safety for tool inputs and outputs on the client and server.

Reading UI Message Streams

`UIMessage` streams are useful outside of traditional chat use cases. You can consume them for terminal UIs, custom stream processing on the client, or React Server Components (RSC).

The `readUIMessageStream` helper transforms a stream of `UIMessageChunk` objects into an `AsyncIterableStream` of `UIMessage` objects, allowing you to process messages as they're being constructed.

Basic Usage

```
import { openai } from '@ai-sdk/openai';

import { readUIMessageStream, streamText } from 'ai';

async function main() {
  const result = streamText({
    model: openai('gpt-4o'),
    prompt: 'Write a short story about a robot.',
  });

  for await (const uiMessage of readUIMessageStream({
    stream: result.toUIMessageStream(),
  })) {
    console.log('Current message state:', uiMessage);
  }
}
```

Tool Calls Integration

Handle streaming responses that include tool calls:

```
import { openai } from '@ai-sdk/openai';

import { readUIMessageStream, streamText, tool } from 'ai';

import { z } from 'zod';

async function handleToolCalls() {
  const result = streamText({
    model: openai('gpt-4o'),
    tools: {
      weather: tool({
        description: 'Get the weather in a location',
        inputSchema: z.object({
          location: z.string().describe('The location to get the weather for'),
        }),
        execute: ({ location }) => ({
          location,
          temperature: 72 + Math.floor(Math.random() * 21) - 10,
        }),
      }),
```

```
    },
    prompt: 'What is the weather in Tokyo?',
  });

  for await (const uiMessage of readUIMessageStream({
    stream: result.toUIMessageStream(),
  })) {
    // Handle different part types

    uiMessage.parts.forEach(part => {
      switch (part.type) {
        case 'text':
          console.log('Text:', part.text);
          break;
        case 'tool-call':
          console.log('Tool called:', part.toolName, 'with args:', part.args);
          break;
        case 'tool-result':
          console.log('Tool result:', part.result);
          break;
      }
    });
  }
}
```

Resuming Conversations

Resume streaming from a previous message state:

```
import { readUIMessageStream, streamText } from 'ai';

async function resumeConversation(lastMessage: UIMessage) {
  const result = streamText({
    model: openai('gpt-4o'),
    messages: [
      { role: 'user', content: 'Continue our previous conversation.' },
    ],
  });

  // Resume from the last message
  for await (const uiMessage of readUIMessageStream({
    stream: result.toUIMessageStream(),
    message: lastMessage, // Resume from this message
  })) {
    console.log('Resumed message:', uiMessage);
  }
}
```

Message Metadata

Message metadata allows you to attach custom information to messages at the message level. This is useful for tracking timestamps, model information, token usage, user context, and other message-level data.

Overview

Message metadata differs from data parts in that it's attached at the message level rather than being part of the message content. While data parts are ideal for dynamic content that forms part of the message, metadata is perfect for information about the message itself.

Getting Started

Here's a simple example of using message metadata to track timestamps and model information:

Defining Metadata Types

First, define your metadata type for type safety:

```
// app/types.ts

import { UIMessage } from 'ai';

import { z } from 'zod';

// Define your metadata schema
export const messageMetadataSchema = z.object({
  createdAt: z.number().optional(),
  model: z.string().optional(),
  totalTokens: z.number().optional(),
});

export type MessageMetadata = z.infer<typeof messageMetadataSchema>;

// Create a typed UIMessage
export type MyUIMessage = UIMessage<MessageMetadata>;
```

Sending Metadata from the Server

Use the `messageMetadata` callback in `toUIMessageStreamResponse` to send metadata at different streaming stages:

```
// app/api/chat/route.ts

import { openai } from '@ai-sdk/openai';

import { convertToModelMessages, streamText } from 'ai';

import { MyUIMessage } from '@/types';

export async function POST(req: Request) {
```

```
  const { messages }: { messages: MyUIMessage[] } = await req.json();

  const result = streamText({
    model: openai('gpt-4o'),
    messages: convertToModelMessages(messages),
  });

  return result.toUIMessageStreamResponse({
    originalMessages: messages, // pass this in for type-safe return objects
    messageMetadata: ({ part }) => {
      // Send metadata when streaming starts
      if (part.type === 'start') {
        return {
          createdAt: Date.now(),
          model: 'gpt-4o',
        };
      }

      // Send additional metadata when streaming completes
      if (part.type === 'finish') {
        return {
          totalTokens: part.totalUsage.totalTokens,
        };
      }
    },
  });
}
```

To enable type-safe metadata return object in `messageMetadata`, pass in the `originalMessages` parameter typed to your UIMessage type.

Accessing Metadata on the Client

Access metadata through the `message.metadata` property:

```
// app/page.tsx
'use client';

import { useChat } from '@ai-sdk/react';

import { DefaultChatTransport } from 'ai';

import { MyUIMessage } from '@/types';

export default function Chat() {
  const { messages } = useChat<MyUIMessage>({
    transport: new DefaultChatTransport({
      api: '/api/chat',
    }),
  });

  return (
    <div>
      {messages.map((message) => (
        <div key={message.id}>
```

```
            <div>
              {message.role === 'user' ? 'User: ' : 'AI: '}
              {message.metadata?.createdAt && (
                <span className="text-sm text-gray-500">
                  {new Date(message.metadata.createdAt).toLocaleTimeString()}
                </span>
              )}
            </div>

            {/* Render message content */}
            {message.parts.map((part, index) =>
              part.type === 'text' ? <div key={index}>{part.text}</div> : null,
            )}

            {/* Display additional metadata */}
            {message.metadata?.totalTokens && (
              <div className="text-xs text-gray-400">
                {message.metadata.totalTokens} tokens
              </div>
            )}
          </div>
        ))}
      </div>
    );
}
```

For streaming arbitrary data that changes during generation, consider using data parts instead.

Common Use Cases

Message metadata is ideal for:

- **Timestamps:** When messages were created or completed
- **Model Information:** Which AI model was used
- **Token Usage:** Track costs and usage limits
- **User Context:** User IDs, session information
- **Performance Metrics:** Generation time, time to first token
- **Quality Indicators:** Finish reason, confidence scores

See Also

- Chatbot Guide - Message metadata in the context of building chatbots
- Streaming Data - Comparison with data parts
- UIMessage Reference - Complete UIMessage type reference

Stream Protocols

AI SDK UI functions such as `useChat` and `useCompletion` support both text streams and data streams.
The stream protocol defines how the data is streamed to the frontend on top of the HTTP protocol.

This page describes both protocols and how to use them in the backend and frontend.

You can use this information to develop custom backends and frontends for your use case, e.g.,
to provide compatible API endpoints that are implemented in a different language such as Python.

For instance, here's an example using FastAPI as a backend.

Text Stream Protocol

A text stream contains chunks in plain text, that are streamed to the frontend.
Each chunk is then appended together to form a full text response.

Text streams are supported by `useChat`, `useCompletion`, and `useObject`.
When you use `useChat` or `useCompletion`, you need to enable text streaming
by setting the `streamProtocol` options to `text`.

You can generate text streams with `streamText` in the backend.
When you call `toTextStreamResponse()` on the result object,
a streaming HTTP response is returned.

> Text streams only support basic text data. If you need to stream other types of data such as tool calls, use data streams.

Text Stream Example

Here is a Next.js example that uses the text stream protocol:

```
'use client';

import { useChat } from '@ai-sdk/react';
import { TextStreamChatTransport } from 'ai';
import { useState } from 'react';

export default function Chat() {
  const [input, setInput] = useState('');
  const { messages, sendMessage } = useChat({
    transport: new TextStreamChatTransport({ api: '/api/chat' }),
  });

  return (
    <div className="flex flex-col w-full max-w-md py-24 mx-auto stretch">
      {messages.map(message => (
        <div key={message.id} className="whitespace-pre-wrap">
          {message.role === 'user' ? 'User: ' : 'AI: '}
          {message.parts.map((part, i) => {
            switch (part.type) {
              case 'text':
```

```
                return <div key={`${message.id}-${i}`}>{part.text}</div>;
              }
            })}
          </div>
        ))}
        <form
          onSubmit={e => {
            e.preventDefault();
            sendMessage({ text: input });
            setInput('');
          }}
        >
          <input
            className="fixed dark:bg-zinc-900 bottom-0 w-full max-w-md p-2 mb-8 bor
            value={input}
            placeholder="Say something..."
            onChange={e => setInput(e.currentTarget.value)}
          />
        </form>
      </div>
    );
}
```

```
import { streamText, UIMessage, convertToModelMessages } from 'ai';
import { openai } from '@ai-sdk/openai';

// Allow streaming responses up to 30 seconds
export const maxDuration = 30;

export async function POST(req: Request) {
  const { messages }: { messages: UIMessage[] } = await req.json();

  const result = streamText({
    model: openai('gpt-4o'),
    messages: convertToModelMessages(messages),
  });

  return result.toTextStreamResponse();
}
```

Data Stream Protocol

A data stream follows a special protocol that the AI SDK provides to send information to the frontend.

The data stream protocol uses Server-Sent Events (SSE) format for improved standardization, keep-alive through ping, reconnect capabilities, and better cache handling.

> When you provide data streams from a custom backend, you need to set the `x-vercel-ai-ui-message-stream` header to `v1`.

The following stream parts are currently supported:

Message Start Part

Indicates the beginning of a new message with metadata.

Format: Server-Sent Event with JSON object

Example:

```
data: {"type":"start","messageId":"..."}
```

Text Parts

Text content is streamed using a start/delta/end pattern with unique IDs for each text block.

Text Start Part

Indicates the beginning of a text block.

Format: Server-Sent Event with JSON object

Example:

```
data: {"type":"text-start","id":"msg_68679a454370819ca74c8eb3d04379630dd1afb72306
```

Text Delta Part

Contains incremental text content for the text block.

Format: Server-Sent Event with JSON object

Example:

```
data: {"type":"text-delta","id":"msg_68679a454370819ca74c8eb3d04379630dd1afb72306
```

Text End Part

Indicates the completion of a text block.

Format: Server-Sent Event with JSON object

Example:

```
data: {"type":"text-end","id":"msg_68679a454370819ca74c8eb3d04379630dd1afb72306ca
```

Reasoning Parts

Reasoning content is streamed using a start/delta/end pattern with unique IDs for each reasoning block.

Reasoning Start Part

Indicates the beginning of a reasoning block.

Format: Server-Sent Event with JSON object

Example:

```
data: {"type":"reasoning-start","id":"reasoning_123"}
```

Reasoning Delta Part

Contains incremental reasoning content for the reasoning block.

Format: Server-Sent Event with JSON object

Example:

```
data: {"type":"reasoning-delta","id":"reasoning_123","delta":"This is some reason
```

Reasoning End Part

Indicates the completion of a reasoning block.

Format: Server-Sent Event with JSON object

Example:

```
data: {"type":"reasoning-end","id":"reasoning_123"}
```

Source Parts

Source parts provide references to external content sources.

Source URL Part

References to external URLs.

Format: Server-Sent Event with JSON object

Example:

```
data: {"type":"source-url","sourceId":"https://example.com","url":"https://exampl
```

Source Document Part

References to documents or files.

Format: Server-Sent Event with JSON object

Example:

```
data: {"type":"source-document","sourceId":"https://example.com","mediaType":"fil
```

File Part

The file parts contain references to files with their media type.

Format: Server-Sent Event with JSON object

Example:

```
data: {"type":"file","url":"https://example.com/file.png","mediaType":"image/png"
```

Data Parts

Custom data parts allow streaming of arbitrary structured data with type-specific handling.

Format: Server-Sent Event with JSON object where the type includes a custom suffix

Example:

```
data: {"type":"data-weather","data":{"location":"SF","temperature":100}}
```

The `data-*` type pattern allows you to define custom data types that your frontend can handle specifically.

Error Part

The error parts are appended to the message as they are received.

Format: Server-Sent Event with JSON object

Example:

```
data: {"type":"error","errorText":"error message"}
```

Tool Input Start Part

Indicates the beginning of tool input streaming.

Format: Server-Sent Event with JSON object

Example:

```
data: {"type":"tool-input-start","toolCallId":"call_fJdQDqnXeGxTmr4E3YPSR7Ar","to
```

Tool Input Delta Part

Incremental chunks of tool input as it's being generated.

Format: Server-Sent Event with JSON object

Example:

```
data: {"type":"tool-input-delta","toolCallId":"call_fJdQDqnXeGxTmr4E3YPSR7Ar","in
```

Tool Input Available Part

Indicates that tool input is complete and ready for execution.

Format: Server-Sent Event with JSON object

Example:

```
data: {"type":"tool-input-available","toolCallId":"call_fJdQDqnXeGxTmr4E3YPSR7Ar"
```

Tool Output Available Part

Contains the result of tool execution.

Format: Server-Sent Event with JSON object

Example:

```
data: {"type":"tool-output-available","toolCallId":"call_fJdQDqnXeGxTmr4E3YPSR7Ar
```

Start Step Part

A part indicating the start of a step.

Format: Server-Sent Event with JSON object

Example:

```
data: {"type":"start-step"}
```

Finish Step Part

A part indicating that a step (i.e., one LLM API call in the backend) has been completed.

This part is necessary to correctly process multiple stitched assistant calls, e.g. when calling tools in the backend, and using steps in `useChat` at the same time.

Format: Server-Sent Event with JSON object

Example:

```
data: {"type":"finish-step"}
```

Finish Message Part

A part indicating the completion of a message.

Format: Server-Sent Event with JSON object

Example:

```
data: {"type":"finish"}
```

Stream Termination

The stream ends with a special `[DONE]` marker.

Format: Server-Sent Event with literal `[DONE]`

Example:

```
data: [DONE]
```

The data stream protocol is supported
by `useChat` and `useCompletion` on the frontend and used by default.
`useCompletion` only supports the `text` and `data` stream parts.

On the backend, you can use `toUIMessageStreamResponse()` from the `streamText` result object to return a streaming HTTP response.

UI Message Stream Example

Here is a Next.js example that uses the UI message stream protocol:

```
'use client';

import { useChat } from '@ai-sdk/react';
import { useState } from 'react';

export default function Chat() {
  const [input, setInput] = useState('');
```

```
  const { messages, sendMessage } = useChat();

  return (
    <div className="flex flex-col w-full max-w-md py-24 mx-auto stretch">
      {messages.map(message => (
        <div key={message.id} className="whitespace-pre-wrap">
          {message.role === 'user' ? 'User: ' : 'AI: '}
          {message.parts.map((part, i) => {
            switch (part.type) {
              case 'text':
                return <div key={`${message.id}-${i}`}>{part.text}</div>;
            }
          })}
        </div>
      ))}
      <form
        onSubmit={e => {
          e.preventDefault();
          sendMessage({ text: input });
          setInput('');
        }}
      >
        <input
          className="fixed dark:bg-zinc-900 bottom-0 w-full max-w-md p-2 mb-8 bor
          value={input}
          placeholder="Say something..."
          onChange={e => setInput(e.currentTarget.value)}
        />
      </form>
    </div>
  );
}
```

```
import { openai } from '@ai-sdk/openai';
import { streamText, UIMessage, convertToModelMessages } from 'ai';

// Allow streaming responses up to 30 seconds
export const maxDuration = 30;

export async function POST(req: Request) {
  const { messages }: { messages: UIMessage[] } = await req.json();

  const result = streamText({
    model: openai('gpt-4o'),
    messages: convertToModelMessages(messages),
  });

  return result.toUIMessageStreamResponse();
}
```

Chatbot Message Persistence

Being able to store and load chat messages is crucial for most AI chatbots.
In this guide, we'll show how to implement message persistence with `useChat` and `streamText`.

> This guide does not cover authorization, error handling, or other real-world considerations.
> It is intended to be a simple example of how to implement message persistence.

Starting a new chat

When the user navigates to the chat page without providing a chat ID,
we need to create a new chat and redirect to the chat page with the new chat ID.

```
// app/chat/page.tsx
import { redirect } from 'next/navigation';

import { createChat } from '@util/chat-store';

export default async function Page() {
  const id = await createChat(); // create a new chat
  redirect(`/chat/${id}`); // redirect to chat page, see below
}
```

Our example chat store implementation uses files to store the chat messages.
In a real-world application, you would use a database or a cloud storage service,
and get the chat ID from the database.
That being said, the function interfaces are designed to be easily replaced with other implementations.

```
// util/chat-store.ts
import { generateId } from 'ai';

import { existsSync, mkdirSync } from 'fs';

import { writeFile } from 'fs/promises';

import path from 'path';

export async function createChat(): Promise<string> {
  const id = generateId(); // generate a unique chat ID
  await writeFile(getChatFile(id), '[]'); // create an empty chat file
  return id;
}

function getChatFile(id: string): string {
  const chatDir = path.join(process.cwd(), '.chats');

  if (!existsSync(chatDir)) mkdirSync(chatDir, { recursive: true });

  return path.join(chatDir, `${id}.json`);
}
```

Loading an existing chat

When the user navigates to the chat page with a chat ID, we need to load the chat messages and display them.

```
// app/chat/[id]/page.tsx
import { loadChat } from '@util/chat-store';

import Chat from '@ui/chat';

export default async function Page(props: { params: Promise<{ id: string }> }) {
  const { id } = await props.params; // get the chat ID from the URL
  const messages = await loadChat(id); // load the chat messages
  return <Chat id={id} initialMessages={messages} />; // display the chat
}
```

The `loadChat` function in our file-based chat store is implemented as follows:

```
// util/chat-store.ts
import { UIMessage } from 'ai';

import { readFile } from 'fs/promises';

export async function loadChat(id: string): Promise<UIMessage[]> {
  return JSON.parse(await readFile(getChatFile(id), 'utf8'));
}

// ... rest of the file
```

The display component is a simple chat component that uses the `useChat` hook to send and receive messages:

```
// ui/chat.tsx
'use client';

import { UIMessage, useChat } from '@ai-sdk/react';

import { DefaultChatTransport } from 'ai';

import { useState } from 'react';

export default function Chat({ id, initialMessages }: { id?: string | undefined;
  const [input, setInput] = useState('');

  const { sendMessage, messages } = useChat({
    id, // use the provided chat ID
    messages: initialMessages, // load initial messages
    transport: new DefaultChatTransport({
      api: '/api/chat',
    }),
  });

  const handleSubmit = (e: React.FormEvent) => {
    e.preventDefault();
    if (input.trim()) {
      sendMessage({ text: input });
      setInput('');
```

```
    }
  };

  // simplified rendering code, extend as needed:
  return (
    <div>
      {messages.map((m) => (
        <div key={m.id}>
          {m.role === 'user' ? 'User: ' : 'AI: '}
          {m.parts
            .map((part) => (part.type === 'text' ? part.text : ''))
            .join('')}
        </div>
      ))}

      <form onSubmit={handleSubmit}>
        <input
          value={input}
          onChange={(e) => setInput(e.target.value)}
          placeholder="Type a message..."
        />
        <button type="submit">Send</button>
      </form>
    </div>
  );
}
```

Storing messages

`useChat` sends the chat id and the messages to the backend.

> The `useChat` message format is different from the `ModelMessage` format. The `useChat` message format is designed for frontend display, and contains additional fields such as `id` and `createdAt`. We recommend storing the messages in the `useChat` message format.

Storing messages is done in the `onFinish` callback of the `toUIMessageStreamResponse` function.
`onFinish` receives the complete messages including the new AI response as `UIMessage[]`.

```
// app/api/chat/route.ts
import { openai } from '@ai-sdk/openai';

import { saveChat } from '@util/chat-store';

import { convertToModelMessages, streamText, UIMessage } from 'ai';

export async function POST(req: Request) {
  const { messages, chatId }: { messages: UIMessage[]; chatId: string } = await r

  const result = streamText({
    model: openai('gpt-4o-mini'),
    messages: convertToModelMessages(messages),
  });
```

```
  return result.toUIMessageStreamResponse({
    originalMessages: messages,
    onFinish: ({ messages }) => {
      saveChat({ chatId, messages });
    },
  });
}
```

The actual storage of the messages is done in the `saveChat` function, which in our file-based chat store is implemented as follows:

```
// util/chat-store.ts
import { UIMessage } from 'ai';

import { writeFile } from 'fs/promises';

export async function saveChat({
  chatId,
  messages,
}: {
  chatId: string;
  messages: UIMessage[];
}): Promise<void> {
  const content = JSON.stringify(messages, null, 2);
  await writeFile(getChatFile(chatId), content);
}

// ... rest of the file
```

Message IDs

In addition to a chat ID, each message has an ID.
You can use this message ID to e.g. manipulate individual messages.

Client-side vs Server-side ID Generation

By default, message IDs are generated client-side:

- User message IDs are generated by the `useChat` hook on the client
- AI response message IDs are generated by `streamText` on the server

For applications without persistence, client-side ID generation works perfectly.
However, **for persistence, you need server-side generated IDs** to ensure consistency across sessions and prevent ID conflicts when messages are stored and retrieved.

Setting Up Server-side ID Generation

When implementing persistence, you have two options for generating server-side IDs:

1. Using `generateMessageId` in `toUIMessageStreamResponse`
2. Setting IDs in your start message part with `createUIMessageStream`

Option 1: Using `generateMessageId` in `toUIMessageStreamResponse`

You can control the ID format by providing ID generators using createIdGenerator():

```
// app/api/chat/route.ts
import { createIdGenerator, streamText } from 'ai';

export async function POST(req: Request) {
  // ...
  const result = streamText({
    // ...
  });

  return result.toUIMessageStreamResponse({
    originalMessages: messages,

    // Generate consistent server-side IDs for persistence:
    generateMessageId: createIdGenerator({
      prefix: 'msg',
      size: 16,
    }),

    onFinish: ({ messages }) => {
      saveChat({ chatId, messages });
    },
  });
}
```

Option 2: Setting IDs with `createUIMessageStream`

Alternatively, you can use `createUIMessageStream` to control the message ID by writing a start message part:

```
// app/api/chat/route.ts
import {
  generateId,
  streamText,
  createUIMessageStream,
  createUIMessageStreamResponse,
} from 'ai';

export async function POST(req: Request) {
  const { messages, chatId } = await req.json();

  const stream = createUIMessageStream({
    execute: ({ writer }) => {
      // Write start message part with custom ID
      writer.write({
        type: 'start',
        messageId: generateId(), // Generate server-side ID for persistence
      });

      const result = streamText({
        model: openai('gpt-4o-mini'),
        messages: convertToModelMessages(messages),
      });
```

```
      writer.merge(result.toUIMessageStream({ sendStart: false })); // omit start
    },
  });

  return createUIMessageStreamResponse({ stream });
}
```

For client-side applications that don't require persistence, you can still customize client-side ID generation:

```
// ui/chat.tsx
import { createIdGenerator } from 'ai';

import { useChat } from '@ai-sdk/react';

const { ... } = useChat({
  generateId: createIdGenerator({
    prefix: 'msgc',
    size: 16,
  }),
  // ...
});
```

Sending only the last message

Once you have implemented message persistence, you might want to send only the last message to the server.
This reduces the amount of data sent to the server on each request and can improve performance.

To achieve this, you can provide a `prepareSendMessagesRequest` function to the transport.
This function receives the messages and the chat ID, and returns the request body to be sent to the server.

```
// ui/chat.tsx
import { useChat } from '@ai-sdk/react';

import { DefaultChatTransport } from 'ai';

const {
  // ...
} = useChat({
  // ...
  transport: new DefaultChatTransport({
    api: '/api/chat',
    // only send the last message to the server:
    prepareSendMessagesRequest({ messages, id }) {
      return { body: { message: messages[messages.length - 1], id } };
    },
  }),
});
```

On the server, you can then load the previous messages and append the new message to the previous messages:

```
// app/api/chat/route.ts
import { convertToModelMessages, UIMessage } from 'ai';

export async function POST(req: Request) {
  // get the last message from the client:
  const { message, id } = await req.json();

  // load the previous messages from the server:
  const previousMessages = await loadChat(id);

  // append the new message to the previous messages:
  const messages = [...previousMessages, message];

  const result = streamText({
    // ...
    messages: convertToModelMessages(messages),
  });

  return result.toUIMessageStreamResponse({
    originalMessages: messages,
    onFinish: ({ messages }) => {
      saveChat({ chatId: id, messages });
    },
  });
}
```

Handling client disconnects

By default, the AI SDK `streamText` function uses backpressure to the language model provider to prevent
the consumption of tokens that are not yet requested.

However, this means that when the client disconnects, e.g. by closing the browser tab or because of a network issue,
the stream from the LLM will be aborted and the conversation may end up in a broken state.

Assuming that you have a storage solution in place, you can use the `consumeStream` method to consume the stream on the backend,
and then save the result as usual.
`consumeStream` effectively removes the backpressure,
meaning that the result is stored even when the client has already disconnected.

```
// app/api/chat/route.ts
import { convertToModelMessages, streamText, UIMessage } from 'ai';

import { saveChat } from '@util/chat-store';

export async function POST(req: Request) {
  const { messages, chatId }: { messages: UIMessage[]; chatId: string } = await r

  const result = streamText({
    model,
    messages: convertToModelMessages(messages),
  });
```

```
  // consume the stream to ensure it runs to completion & triggers onFinish
  // even when the client response is aborted:
  result.consumeStream(); // no await

  return result.toUIMessageStreamResponse({
    originalMessages: messages,
    onFinish: ({ messages }) => {
      saveChat({ chatId, messages });
    },
  });
}
```

When the client reloads the page after a disconnect, the chat will be restored from the storage solution.

> In production applications, you would also track the state of the request (in progress, complete) in your stored messages and use it on the client to cover the case where the client reloads the page after a disconnection, but the streaming is not yet complete.

Resuming ongoing streams

> This feature is experimental and may change in future versions.

The `useChat` hook has experimental support for resuming an ongoing chat generation stream by any client, either after a network disconnect or by reloading the chat page. This can be useful for building applications that involve long-running conversations or for ensuring that messages are not lost in case of network failures.

The following are the pre-requisities for your chat application to support resumable streams:

- Installing the resumable-stream package that helps create and manage the publisher/subscriber mechanism of the streams.
- Creating a Redis instance to store the stream state.
- Creating a table that tracks the stream IDs associated with a chat.

To resume a chat stream, you will use the `resumeStream` function returned by the `useChat` hook. You will call this function during the initial mount of the hook inside the main chat component.

```
// ui/chat.tsx
'use client';

import { useChat } from '@ai-sdk/react';

import { DefaultChatTransport, type UIMessage } from 'ai';

import { useEffect } from 'react';

export function Chat({ chatId, autoResume, initialMessages = [] }: { chatId: stri
  const { resumeStream, /* ... other useChat returns */ } = useChat({
    id: chatId,
    messages: initialMessages,
    transport: new DefaultChatTransport({
      api: '/api/chat',
    }),
```

```
  });

  useEffect(() => {
    if (autoResume) {
      resumeStream();
    }

    // We want to disable the exhaustive deps rule here because we only want to r
    // eslint-disable-next-line react-hooks/exhaustive-deps
  }, []);

  return <div>{/* Your chat UI here */}</div>;
}
```

The `resumeStream` function makes a `GET` request to your configured chat endpoint (or `/api/chat` by default) whenever your client calls it. If there's an active stream, it will pick up where it left off, otherwise it simply finishes without error.

The `GET` request automatically appends the `chatId` query parameter to the URL to help identify the chat the request belongs to. Using the `chatId`, you can look up the most recent stream ID from the database and resume the stream.

```
GET /api/chat?chatId=<your-chat-id>
```

Earlier, you must've implemented the `POST` handler for the `/api/chat` route to create new chat generations. When using `resumeStream`, you must also implement the `GET` handler for `/api/chat` route to resume streams.

1. Implement the GET handler

Add a `GET` method to `/api/chat` that:

1. Reads `chatId` from the query string
2. Validates it's present
3. Loads any stored stream IDs for that chat
4. Returns the latest one to `streamContext.resumableStream()`
5. Falls back to an empty stream if it's already closed

```
// app/api/chat/route.ts
import { loadStreams } from '@util/chat-store';

import { createUIMessageStream, JsonToSseTransformStream } from 'ai';

import { after } from 'next/server';

import { createResumableStreamContext } from 'resumable-stream';

export async function GET(request: Request) {
  const streamContext = createResumableStreamContext({
    waitUntil: after,
  });

  const { searchParams } = new URL(request.url);
  const chatId = searchParams.get('chatId');
```

```
  if (!chatId) {
    return new Response('id is required', { status: 400 });
  }

  const streamIds = await loadStreams(chatId);

  if (!streamIds.length) {
    return new Response('No streams found', { status: 404 });
  }

  const recentStreamId = streamIds.at(-1);

  if (!recentStreamId) {
    return new Response('No recent stream found', { status: 404 });
  }

  const emptyDataStream = createUIMessageStream({
    execute: () => {},
  });

  return new Response(
    await streamContext.resumableStream(recentStreamId, () =>
      emptyDataStream.pipeThrough(new JsonToSseTransformStream()),
    ),
  );
}
```

After you've implemented the `GET` handler, you can update the `POST` handler to handle the creation of resumable streams.

2. Update the POST handler

When you create a brand-new chat completion, you must:

1. Generate a fresh `streamId`
2. Persist it alongside your `chatId`
3. Kick off a `createUIMessageStream` that pipes tokens as they arrive
4. Hand that new stream to `streamContext.resumableStream()`

```
// app/api/chat/route.ts
import {
  convertToModelMessages,
  createUIMessageStream,
  generateId,
  streamText,
  UIMessage,
} from 'ai';

import { appendStreamId, saveChat } from '@util/chat-store';

import { createResumableStreamContext } from 'resumable-stream';

import { openai } from '@ai-sdk/openai';

const streamContext = createResumableStreamContext({
```

```
  waitUntil: after,
});

async function POST(request: Request) {
  const { chatId, messages }: { chatId: string; messages: UIMessage[] } = await r

  const streamId = generateId();

  // Record this new stream so we can resume later
  await appendStreamId({ chatId, streamId });

  // Build the data stream that will emit tokens
  const stream = createUIMessageStream({
    execute: ({ writer }) => {
      const result = streamText({
        model: openai('gpt-4o'),
        messages: convertToModelMessages(messages),
      });

      // Return a resumable stream to the client
      writer.merge(result.toUIMessageStream());
    },
  });

  const resumableStream = await streamContext.resumableStream(streamId, () => str

  return resumableStream.toUIMessageStreamResponse({
    originalMessages: messages,
    onFinish: ({ messages }) => {
      saveChat({ chatId, messages });
    },
  });
}
```

With both handlers, your clients can now gracefully resume ongoing streams.

Chatbot Tool Usage

With `useChat` and `streamText`, you can use tools in your chatbot application.
The AI SDK supports three types of tools in this context:

1. Automatically executed server-side tools
2. Automatically executed client-side tools
3. Tools that require user interaction, such as confirmation dialogs

The flow is as follows:

1. The user enters a message in the chat UI.
2. The message is sent to the API route.
3. In your server side route, the language model generates tool calls during the `streamText` call.
4. All tool calls are forwarded to the client.
5. Server-side tools are executed using their `execute` method and their results are forwarded to the client.
6. Client-side tools that should be automatically executed are handled with the `onToolCall` callback.
   You must call `addToolResult` to provide the tool result.
7. Client-side tool that require user interactions can be displayed in the UI.
   The tool calls and results are available as tool invocation parts in the `parts` property of the last assistant message.
8. When the user interaction is done, `addToolResult` can be used to add the tool result to the chat.
9. The chat can be configured to automatically submit when all tool results are available using `sendAutomaticallyWhen`.
   This triggers another iteration of this flow.

The tool calls and tool executions are integrated into the assistant message as typed tool parts.
A tool part is at first a tool call, and then it becomes a tool result when the tool is executed.
The tool result contains all information about the tool call as well as the result of the tool execution.

Tool result submission can be configured using the `sendAutomaticallyWhen` option.
You can use the `lastAssistantMessageIsCompleteWithToolCalls` helper to automatically submit when all tool results are available.
This simplifies the client-side code while still allowing full control when needed.

Example

In this example, we'll use three tools:

- `getWeatherInformation`: An automatically executed server-side tool that returns the weather in a given city.
- `askForConfirmation`: A user-interaction client-side tool that asks the user for confirmation.
- `getLocation`: An automatically executed client-side tool that returns a random city.

API route

```
// app/api/chat/route.ts

import { openai } from '@ai-sdk/openai';
```

```
import { convertToModelMessages, streamText, UIMessage } from 'ai';

import { z } from 'zod';

// Allow streaming responses up to 30 seconds

export const maxDuration = 30;

export async function POST(req: Request) {
  const { messages }: { messages: UIMessage[] } = await req.json();

  const result = streamText({
    model: openai('gpt-4o'),
    messages: convertToModelMessages(messages),
    tools: {
      // server-side tool with execute function:
      getWeatherInformation: {
        description: 'show the weather in a given city to the user',
        inputSchema: z.object({ city: z.string() }),
        execute: async ({ city }: { city: string }) => {
          const weatherOptions = ['sunny', 'cloudy', 'rainy', 'snowy', 'windy'];
          return weatherOptions[
            Math.floor(Math.random() * weatherOptions.length)
          ];
        },
      },

      // client-side tool that starts user interaction:
      askForConfirmation: {
        description: 'Ask the user for confirmation.',
        inputSchema: z.object({
          message: z.string().describe('The message to ask for confirmation.'),
        }),
      },

      // client-side tool that is automatically executed on the client:
      getLocation: {
        description:
          'Get the user location. Always ask for confirmation before using this t
        inputSchema: z.object({}),
      },
    },
  });

  return result.toUIMessageStreamResponse();
}
```

Client-side page

The client-side page uses the `useChat` hook to create a chatbot application with real-time message streaming.
Tool calls are displayed in the chat UI as typed tool parts.
Please make sure to render the messages using the `parts` property of the message.

There are three things worth mentioning:

1. The `onToolCall` callback is used to handle client-side tools that should be automatically executed.
   In this example, the `getLocation` tool is a client-side tool that returns a random city.
   You call `addToolResult` to provide the result (without `await` to avoid potential deadlocks).
2. The `sendAutomaticallyWhen` option with `lastAssistantMessageIsCompleteWithToolCalls` helper automatically submits when all tool results are available.
3. The `parts` array of assistant messages contains tool parts with typed names like `tool-askForConfirmation`.
   The client-side tool `askForConfirmation` is displayed in the UI.
   It asks the user for confirmation and displays the result once the user confirms or denies the execution.
   The result is added to the chat using `addToolResult` with the `tool` parameter for type safety.

```
// app/page.tsx
'use client';

import { useChat } from '@ai-sdk/react';
import {
  DefaultChatTransport,
  lastAssistantMessageIsCompleteWithToolCalls,
} from 'ai';
import { useState } from 'react';

export default function Chat() {
  const { messages, sendMessage, addToolResult } = useChat({
    transport: new DefaultChatTransport({
      api: '/api/chat',
    }),

    sendAutomaticallyWhen: lastAssistantMessageIsCompleteWithToolCalls,

    // run client-side tools that are automatically executed:
    async onToolCall({ toolCall }) {
      if (toolCall.toolName === 'getLocation') {
        const cities = ['New York', 'Los Angeles', 'Chicago', 'San Francisco'];

        // No await - avoids potential deadlocks
        addToolResult({
          tool: 'getLocation',
          toolCallId: toolCall.toolCallId,
          output: cities[Math.floor(Math.random() * cities.length)],
        });
      }
    },
  });

  const [input, setInput] = useState('');

  return (
    <>
```

```
{messages?.map((message) => (
  <div key={message.id}>
    <strong>{`${message.role}: `}</strong>
    {message.parts.map((part) => {
      switch (part.type) {
        // render text parts as simple text:
        case 'text':
          return part.text;

        // for tool parts, use the typed tool part names:
        case 'tool-askForConfirmation': {
          const callId = part.toolCallId;

          switch (part.state) {
            case 'input-streaming':
              return <div key={callId}>Loading confirmation request...</div

            case 'input-available':
              return (
                <div key={callId}>
                  {part.input.message}
                  <div>
                    <button
                      onClick={() =>
                        addToolResult({
                          tool: 'askForConfirmation',
                          toolCallId: callId,
                          output: 'Yes, confirmed.',
                        })
                      }
                    >
                      Yes
                    </button>
                    <button
                      onClick={() =>
                        addToolResult({
                          tool: 'askForConfirmation',
                          toolCallId: callId,
                          output: 'No, denied',
                        })
                      }
                    >
                      No
                    </button>
                  </div>
                </div>
              );

            case 'output-available':
              return <div key={callId}>Location access allowed: {part.outpu

            case 'output-error':
              return <div key={callId}>Error: {part.errorText}</div>;
          }
```

```
      break;
    }

    case 'tool-getLocation': {
      const callId = part.toolCallId;

      switch (part.state) {
        case 'input-streaming':
          return <div key={callId}>Preparing location request...</div>;

        case 'input-available':
          return <div key={callId}>Getting location...</div>;

        case 'output-available':
          return <div key={callId}>Location: {part.output}</div>;

        case 'output-error':
          return <div key={callId}>Error getting location: {part.errorT
      }

      break;
    }

    case 'tool-getWeatherInformation': {
      const callId = part.toolCallId;

      switch (part.state) {
        // example of pre-rendering streaming tool inputs:
        case 'input-streaming':
          return <pre key={callId}>{JSON.stringify(part, null, 2)}</pre

        case 'input-available':
          return <div key={callId}>Getting weather information for {par

        case 'output-available':
          return (
            <div key={callId}>
              Weather in {part.input.city}: {part.output}
            </div>
          );

        case 'output-error':
          return (
            <div key={callId}>
              Error getting weather for {part.input.city}: {part.errorT
            </div>
          );
      }

      break;
    }
  }
})}
```

```
          <br />
        </div>
      ))}

      <form
        onSubmit={(e) => {
          e.preventDefault();
          if (input.trim()) {
            sendMessage({ text: input });
            setInput('');
          }
        }}
      >
        <input value={input} onChange={(e) => setInput(e.target.value)} />
      </form>
    </>
  );
}
```

Dynamic Tools

When using dynamic tools (tools with unknown types at compile time), the UI parts use a generic `dynamic-tool` type instead of specific tool types:

```
// app/page.tsx

message.parts.map((part, index) => {
  switch (part.type) {
    // Static tools with specific (`tool-${toolName}`) types
    case 'tool-getWeatherInformation':
      return <WeatherDisplay part={part} />;

    // Dynamic tools use generic `dynamic-tool` type
    case 'dynamic-tool':
      return (
        <div key={index}>
          <h4>Tool: {part.toolName}</h4>
          {part.state === 'input-streaming' && (
            <pre>{JSON.stringify(part.input, null, 2)}</pre>
          )}
          {part.state === 'output-available' && (
            <pre>{JSON.stringify(part.output, null, 2)}</pre>
          )}
          {part.state === 'output-error' && <div>Error: {part.errorText}</div>}
        </div>
      );
  }
});
```

Dynamic tools are useful when integrating with:

- MCP (Model Context Protocol) tools without schemas
- User-defined functions loaded at runtime
- External tool providers

Tool call streaming

Tool call streaming is **enabled by default** in AI SDK 5.0, allowing you to stream tool calls while they are being generated.
This provides a better user experience by showing tool inputs as they are generated in real-time.

```
// app/api/chat/route.ts

export async function POST(req: Request) {
  const { messages }: { messages: UIMessage[] } = await req.json();

  const result = streamText({
    model: openai('gpt-4o'),
    messages: convertToModelMessages(messages),
    // toolCallStreaming is enabled by default in v5
    // ...
  });

  return result.toUIMessageStreamResponse();
}
```

With tool call streaming enabled, partial tool calls are streamed as part of the data stream.
They are available through the `useChat` hook.
The typed tool parts of assistant messages will also contain partial tool calls.
You can use the `state` property of the tool part to render the correct UI.

```
// app/page.tsx

export default function Chat() {
  // ...

  return (
    <>
      {messages?.map((message) => (
        <div key={message.id}>
          {message.parts.map((part) => {
            switch (part.type) {
              case 'tool-askForConfirmation':
              case 'tool-getLocation':
              case 'tool-getWeatherInformation':
                switch (part.state) {
                  case 'input-streaming':
                    return <pre>{JSON.stringify(part.input, null, 2)}</pre>;

                  case 'input-available':
                    return <pre>{JSON.stringify(part.input, null, 2)}</pre>;

                  case 'output-available':
                    return <pre>{JSON.stringify(part.output, null, 2)}</pre>;

                  case 'output-error':
                    return <div>Error: {part.errorText}</div>;
                }
            }
          })}
```

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

Step start parts

When you are using multi-step tool calls, the AI SDK will add step start parts to the assistant messages.
If you want to display boundaries between tool calls, you can use the `step-start` parts as follows:

```
// app/page.tsx

// ...

// where you render the message parts:
message.parts.map((part, index) => {
  switch (part.type) {
    case 'step-start':
      // show step boundaries as horizontal lines:
      return index > 0 ? (
        <div key={index} className="text-gray-500">
          <hr className="my-2 border-gray-300" />
        </div>
      ) : null;

    case 'text':
      // ...
    case 'tool-askForConfirmation':
    case 'tool-getLocation':
    case 'tool-getWeatherInformation':
      // ...
  }
});

// ...
```

Server-side Multi-Step Calls

You can also use multi-step calls on the server-side with `streamText`.
This works when all invoked tools have an `execute` function on the server side.

```
// app/api/chat/route.ts

import { openai } from '@ai-sdk/openai';

import {
  convertToModelMessages,
  streamText,
  UIMessage,
  stepCountIs,
} from 'ai';

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

```
export async function POST(req: Request) {
  const { messages }: { messages: UIMessage[] } = await req.json();

  const result = streamText({
    model: openai('gpt-4o'),
    messages: convertToModelMessages(messages),
    tools: {
      getWeatherInformation: {
        description: 'show the weather in a given city to the user',
        inputSchema: z.object({ city: z.string() }),
        // tool has execute function:
        execute: async ({ city }: { city: string }) => {
          const weatherOptions = ['sunny', 'cloudy', 'rainy', 'snowy', 'windy'];
          return weatherOptions[
            Math.floor(Math.random() * weatherOptions.length)
          ];
        },
      },
    },
    stopWhen: stepCountIs(5),
  });

  return result.toUIMessageStreamResponse();
}
```

Errors

Language models can make errors when calling tools.
By default, these errors are masked for security reasons, and show up as "An error occurred" in the UI.

To surface the errors, you can use the `onError` function when calling `toUIMessageResponse`.

```
export function errorHandler(error: unknown) {
  if (error == null) {
    return 'unknown error';
  }

  if (typeof error === 'string') {
    return error;
  }

  if (error instanceof Error) {
    return error.message;
  }

  return JSON.stringify(error);
}

const result = streamText({
  // ...
});

return result.toUIMessageStreamResponse({
```

```
  onError: errorHandler,
});
```

In case you are using `createUIMessageResponse`, you can use the `onError` function when calling `toUIMessageResponse`:

```
const response = createUIMessageResponse({
  // ...
  async execute(dataStream) {
    // ...
  },
  onError: (error) => `Custom error: ${error.message}`,
});
```

Generative User Interfaces

Generative user interfaces (generative UI) is the process of allowing a large language model (LLM) to go beyond text and "generate UI". This creates a more engaging and AI-native experience for users.

```
What is the weather in SF?

getWeather("San Francisco")

Thursday, March 7
47°
sunny

7am    48°
8am    50°
9am    52°
10am   54°
11am   56°
12pm   58°
1pm    60°
```

At the core of generative UI are tools, which are functions you provide to the model to perform specialized tasks like getting the weather in a location. The model can decide when and how to use these tools based on the context of the conversation.

Generative UI is the process of connecting the results of a tool call to a React component. Here's how it works:

1. You provide the model with a prompt or conversation history, along with a set of tools.
2. Based on the context, the model may decide to call a tool.
3. If a tool is called, it will execute and return data.
4. This data can then be passed to a React component for rendering.

By passing the tool results to React components, you can create a generative UI experience that's more engaging and adaptive to your needs.

Build a Generative UI Chat Interface

Let's create a chat interface that handles text-based conversations and incorporates dynamic UI elements based on model responses.

Basic Chat Implementation

Start with a basic chat implementation using the `useChat` hook:

```
'use client';

import { useChat } from '@ai-sdk/react';
import { useState } from 'react';

export default function Page() {
  const [input, setInput] = useState('');
  const { messages, sendMessage } = useChat();
```

```
  const handleSubmit = (e: React.FormEvent) => {
    e.preventDefault();
    sendMessage({ text: input });
    setInput('');
  };

  return (
    <div>
      {messages.map(message => (
        <div key={message.id}>
          <div>{message.role === 'user' ? 'User: ' : 'AI: '}</div>
          <div>
            {message.parts.map((part, index) => {
              if (part.type === 'text') {
                return <span key={index}>{part.text}</span>;
              }
              return null;
            })}
          </div>
        </div>
      ))}
      <form onSubmit={handleSubmit}>
        <input
          value={input}
          onChange={e => setInput(e.target.value)}
          placeholder="Type a message..."
        />
        <button type="submit">Send</button>
      </form>
    </div>
  );
}
```

To handle the chat requests and model responses, set up an API route:

```
import { openai } from '@ai-sdk/openai';
import { streamText, convertToModelMessages, UIMessage, stepCountIs } from 'ai';

export async function POST(request: Request) {
  const { messages }: { messages: UIMessage[] } = await request.json();

  const result = streamText({
    model: openai('gpt-4o'),
    system: 'You are a friendly assistant!',
    messages: convertToModelMessages(messages),
    stopWhen: stepCountIs(5),
  });

  return result.toUIMessageStreamResponse();
}
```

This API route uses the `streamText` function to process chat messages and stream the model's responses back to the client.

Create a Tool

Before enhancing your chat interface with dynamic UI elements, you need to create a tool and corresponding React component. A tool will allow the model to perform a specific action, such as fetching weather information.

Create a new file called `ai/tools.ts` with the following content:

```
import { tool as createTool } from 'ai';
import { z } from 'zod';

export const weatherTool = createTool({
  description: 'Display the weather for a location',
  inputSchema: z.object({
    location: z.string().describe('The location to get the weather for'),
  }),
  execute: async function ({ location }) {
    await new Promise(resolve => setTimeout(resolve, 2000));
    return { weather: 'Sunny', temperature: 75, location };
  },
});

export const tools = {
  displayWeather: weatherTool,
};
```

In this file, you've created a tool called `weatherTool`. This tool simulates fetching weather information for a given location. This tool will return simulated data after a 2-second delay. In a real-world application, you would replace this simulation with an actual API call to a weather service.

Update the API Route

Update the API route to include the tool you've defined:

```
import { openai } from '@ai-sdk/openai';
import { streamText, convertToModelMessages, UIMessage, stepCountIs } from 'ai';
import { tools } from '@/ai/tools';

export async function POST(request: Request) {
  const { messages }: { messages: UIMessage[] } = await request.json();

  const result = streamText({
    model: openai('gpt-4o'),
    system: 'You are a friendly assistant!',
    messages: convertToModelMessages(messages),
    stopWhen: stepCountIs(5),
    tools,
  });

  return result.toUIMessageStreamResponse();
}
```

Now that you've defined the tool and added it to your `streamText` call, let's build a React component to display the weather information it returns.

Create UI Components

Create a new file called `components/weather.tsx`:

```
type WeatherProps = {
  temperature: number;
  weather: string;
  location: string;
};

export const Weather = ({ temperature, weather, location }: WeatherProps) => {
  return (
    <div>
      <h2>Current Weather for {location}</h2>
      <p>Condition: {weather}</p>
      <p>Temperature: {temperature}°C</p>
    </div>
  );
};
```

This component will display the weather information for a given location. It takes three props: `temperature`, `weather`, and `location` (exactly what the `weatherTool` returns).

Render the Weather Component

Now that you have your tool and corresponding React component, let's integrate them into your chat interface. You'll render the Weather component when the model calls the weather tool.

To check if the model has called a tool, you can check the `parts` array of the UIMessage object for tool-specific parts. In AI SDK 5.0, tool parts use typed naming: `tool-${toolName}` instead of generic types.

Update your `page.tsx` file:

```
'use client';

import { useChat } from '@ai-sdk/react';
import { useState } from 'react';
import { Weather } from '@/components/weather';

export default function Page() {
  const [input, setInput] = useState('');
  const { messages, sendMessage } = useChat();

  const handleSubmit = (e: React.FormEvent) => {
    e.preventDefault();
    sendMessage({ text: input });
    setInput('');
  };

  return (
    <div>
      {messages.map(message => (
        <div key={message.id}>
          <div>{message.role === 'user' ? 'User: ' : 'AI: '}</div>
```

```
          <div>
            {message.parts.map((part, index) => {
              if (part.type === 'text') {
                return <span key={index}>{part.text}</span>;
              }
              if (part.type === 'tool-displayWeather') {
                switch (part.state) {
                  case 'input-available':
                    return <div key={index}>Loading weather...</div>;
                  case 'output-available':
                    return (
                      <div key={index}>
                        <Weather {...part.output} />
                      </div>
                    );
                  case 'output-error':
                    return <div key={index}>Error: {part.errorText}</div>;
                  default:
                    return null;
                }
              }
              return null;
            })}
          </div>
        </div>
      ))}
      <form onSubmit={handleSubmit}>
        <input
          value={input}
          onChange={e => setInput(e.target.value)}
          placeholder="Type a message..."
        />
        <button type="submit">Send</button>
      </form>
    </div>
  );
}
```

In this updated code snippet, you:

1. Use manual input state management with `useState` instead of the built-in `input` and `handleInputChange`.
2. Use `sendMessage` instead of `handleSubmit` to send messages.
3. Check the `parts` array of each message for different content types.
4. Handle tool parts with type `tool-displayWeather` and their different states (`input-available` , `output-available` , `output-error`).

This approach allows you to dynamically render UI components based on the model's responses, creating a more interactive and context-aware chat experience.

Expanding Your Generative UI Application

You can enhance your chat application by adding more tools and components, creating a richer and more versatile user experience. Here's how you can expand your application:

Adding More Tools

To add more tools, simply define them in your `ai/tools.ts` file:

```
// Add a new stock tool

export const stockTool = createTool({
  description: 'Get price for a stock',
  inputSchema: z.object({
    symbol: z.string().describe('The stock symbol to get the price for'),
  }),
  execute: async function ({ symbol }) {
    // Simulated API call
    await new Promise(resolve => setTimeout(resolve, 2000));
    return { symbol, price: 100 };
  },
});

// Update the tools object

export const tools = {
  displayWeather: weatherTool,
  getStockPrice: stockTool,
};
```

Now, create a new file called `components/stock.tsx`:

```
type StockProps = {
  price: number;
  symbol: string;
};

export const Stock = ({ price, symbol }: StockProps) => {
  return (
    <div>
      <h2>Stock Information</h2>
      <p>Symbol: {symbol}</p>
      <p>Price: ${price}</p>
    </div>
  );
};
```

Finally, update your `page.tsx` file to include the new Stock component:

```
'use client';

import { useChat } from '@ai-sdk/react';
import { useState } from 'react';
import { Weather } from '@/components/weather';
import { Stock } from '@/components/stock';

export default function Page() {
  const [input, setInput] = useState('');
  const { messages, sendMessage } = useChat();
```

```
const handleSubmit = (e: React.FormEvent) => {
  e.preventDefault();
  sendMessage({ text: input });
  setInput('');
};

return (
  <div>
    {messages.map(message => (
      <div key={message.id}>
        <div>{message.role}</div>
        <div>
          {message.parts.map((part, index) => {
            if (part.type === 'text') {
              return <span key={index}>{part.text}</span>;
            }
            if (part.type === 'tool-displayWeather') {
              switch (part.state) {
                case 'input-available':
                  return <div key={index}>Loading weather...</div>;
                case 'output-available':
                  return (
                    <div key={index}>
                      <Weather {...part.output} />
                    </div>
                  );
                case 'output-error':
                  return <div key={index}>Error: {part.errorText}</div>;
                default:
                  return null;
              }
            }
            if (part.type === 'tool-getStockPrice') {
              switch (part.state) {
                case 'input-available':
                  return <div key={index}>Loading stock price...</div>;
                case 'output-available':
                  return (
                    <div key={index}>
                      <Stock {...part.output} />
                    </div>
                  );
                case 'output-error':
                  return <div key={index}>Error: {part.errorText}</div>;
                default:
                  return null;
              }
            }
            return null;
          })}
        </div>
      </div>
    ))}
    <form onSubmit={handleSubmit}>
```

```
        <input
          type="text"
          value={input}
          onChange={e => setInput(e.target.value)}
          placeholder="Type a message..."
        />
        <button type="submit">Send</button>
      </form>
    </div>
  );
}
```

By following this pattern, you can continue to add more tools and components, expanding the capabilities of your Generative UI application.

Completion

The `useCompletion` hook allows you to create a user interface to handle text completions in your application. It enables the streaming of text completions from your AI provider, manages the state for chat input, and updates the UI automatically as new messages are received.

The `useCompletion` hook is now part of the `@ai-sdk/react` package.

In this guide, you will learn how to use the `useCompletion` hook in your application to generate text completions and stream them in real-time to your users.

Example

```
'use client';

import { useCompletion } from '@ai-sdk/react';

export default function Page() {
  const { completion, input, handleInputChange, handleSubmit } = useCompletion({
    api: '/api/completion',
  });

  return (
    <form onSubmit={handleSubmit}>
      <input
        name="prompt"
        value={input}
        onChange={handleInputChange}
        id="input"
      />
      <button type="submit">Submit</button>
      <div>{completion}</div>
    </form>
  );
}
```

```
import { streamText } from 'ai';
import { openai } from '@ai-sdk/openai';

// Allow streaming responses up to 30 seconds
export const maxDuration = 30;

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

  const result = streamText({
    model: openai('gpt-3.5-turbo'),
    prompt,
  });

  return result.toUIMessageStreamResponse();
}
```

In the `Page` component, the `useCompletion` hook will request to your AI provider endpoint whenever the user submits a message. The completion is then streamed back in real-time and displayed in the UI.

This enables a seamless text completion experience where the user can see the AI response as soon as it is available, without having to wait for the entire response to be received.

Customized UI

`useCompletion` also provides ways to manage the prompt via code, show loading and error states, and update messages without being triggered by user interactions.

Loading and error states

To show a loading spinner while the chatbot is processing the user's message, you can use the `isLoading` state returned by the `useCompletion` hook:

```
const { isLoading, ... } = useCompletion();

return (
  <>
    {isLoading ? <Spinner /> : null}
  </>
);
```

Similarly, the `error` state reflects the error object thrown during the fetch request. It can be used to display an error message, or show a toast notification:

```
const { error, ... } = useCompletion();

useEffect(() => {
  if (error) {
    toast.error(error.message);
  }
}, [error]);

// Or display the error message in the UI:
return (
  <>
    {error ? <div>{error.message}</div> : null}
  </>
);
```

Controlled input

In the initial example, we have `handleSubmit` and `handleInputChange` callbacks that manage the input changes and form submissions. These are handy for common use cases, but you can also use uncontrolled APIs for more advanced scenarios such as form validation or customized components.

The following example demonstrates how to use more granular APIs like `setInput` with your custom input and submit button components:

```
const { input, setInput } = useCompletion();

return (
```

```
  <>
    <MyCustomInput
      value={input}
      onChange={value => setInput(value)}
    />
  </>
);
```

Cancelation

It's also a common use case to abort the response message while it's still streaming back from the AI provider. You can do this by calling the `stop` function returned by the `useCompletion` hook.

```
const { stop, isLoading, ... } = useCompletion();

return (
  <>
    <button onClick={stop} disabled={!isLoading}>Stop</button>
  </>
);
```

When the user clicks the "Stop" button, the fetch request will be aborted. This avoids consuming unnecessary resources and improves the UX of your application.

Throttling UI Updates

This feature is currently only available for React.

By default, the `useCompletion` hook will trigger a render every time a new chunk is received. You can throttle the UI updates with the `experimental_throttle` option.

```
const { completion, ... } = useCompletion({
  // Throttle the completion and data updates to 50ms:
  experimental_throttle: 50
});
```

Event Callbacks

`useCompletion` also provides optional event callbacks that you can use to handle different stages of the chatbot lifecycle. These callbacks can be used to trigger additional actions, such as logging, analytics, or custom UI updates.

```
const { ... } = useCompletion({
  onResponse: (response: Response) => {
    console.log('Received response from server:', response);
  },
  onFinish: (prompt: string, completion: string) => {
    console.log('Finished streaming completion:', completion);
  },
  onError: (error: Error) => {
    console.error('An error occurred:', error);
  },
});
```

It's worth noting that you can abort the processing by throwing an error in the `onResponse` callback. This will trigger the `onError` callback and stop the message from being appended to the chat UI. This can be useful for handling unexpected responses from the AI provider.

Configure Request Options

By default, the `useCompletion` hook sends a HTTP POST request to the `/api/completion` endpoint with the prompt as part of the request body. You can customize the request by passing additional options to the `useCompletion` hook:

```
const { messages, input, handleInputChange, handleSubmit } = useCompletion({
  api: '/api/custom-completion',
  headers: {
    Authorization: 'your_token',
  },
  body: {
    user_id: '123',
  },
  credentials: 'same-origin',
});
```

In this example, the `useCompletion` hook sends a POST request to the `/api/completion` endpoint with the specified headers, additional body fields, and credentials for that fetch request. On your server side, you can handle the request with these additional information.

Object Generation

`useObject` is an experimental feature and only available in React.

The `useObject` hook allows you to create interfaces that represent a structured JSON object that is being streamed.

In this guide, you will learn how to use the `useObject` hook in your application to generate UIs for structured data on the fly.

Example

The example shows a small notifications demo app that generates fake notifications in real-time.

Schema

It is helpful to set up the schema in a separate file that is imported on both the client and server.

`app/api/notifications/schema.ts`

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

// define a schema for the notifications

export const notificationSchema = z.object({
  notifications: z.array(
    z.object({
      name: z.string().describe('Name of a fictional person.'),
      message: z.string().describe('Message. Do not use emojis or links.'),
    }),
  ),
});
```

Client

The client uses `useObject` to stream the object generation process.

The results are partial and are displayed as they are received.
Please note the code for handling `undefined` values in the JSX.

`app/page.tsx`

```
'use client';

import { experimental_useObject as useObject } from '@ai-sdk/react';
import { notificationSchema } from './api/notifications/schema';

export default function Page() {
  const { object, submit } = useObject({
    api: '/api/notifications',
    schema: notificationSchema,
  });
```

```
  return (
    <>
      <button onClick={() => submit('Messages during finals week.')}>
        Generate notifications
      </button>

      {object?.notifications?.map((notification, index) => (
        <div key={index}>
          <p>{notification?.name}</p>
          <p>{notification?.message}</p>
        </div>
      ))}
    </>
  );
}
```

Server

On the server, we use `streamObject` to stream the object generation process.

`app/api/notifications/route.ts`

```
import { openai } from '@ai-sdk/openai';
import { streamObject } from 'ai';
import { notificationSchema } from './schema';

// Allow streaming responses up to 30 seconds
export const maxDuration = 30;

export async function POST(req: Request) {
  const context = await req.json();

  const result = streamObject({
    model: openai('gpt-4.1'),
    schema: notificationSchema,
    prompt: `Generate 3 notifications for a messages app in this context:` + cont
  });

  return result.toTextStreamResponse();
}
```

Enum Output Mode

When you need to classify or categorize input into predefined options, you can use the `enum` output mode with `useObject`. This requires a specific schema structure where the object has `enum` as a key with `z.enum` containing your possible values.

Example: Text Classification

This example shows how to build a simple text classifier that categorizes statements as true or false.

Client

When using `useObject` with enum output mode, your schema must be an object with `enum` as the key:

`app/classify/page.tsx`

```
'use client';

import { experimental_useObject as useObject } from '@ai-sdk/react';
import { z } from 'zod';

export default function ClassifyPage() {
  const { object, submit, isLoading } = useObject({
    api: '/api/classify',
    schema: z.object({ enum: z.enum(['true', 'false']) }),
  });

  return (
    <>
      <button onClick={() => submit('The earth is flat')} disabled={isLoading}>
        Classify statement
      </button>

      {object && <div>Classification: {object.enum}</div>}
    </>
  );
}
```

Server

On the server, use `streamObject` with `output: 'enum'` to stream the classification result:

`app/api/classify/route.ts`

```
import { openai } from '@ai-sdk/openai';
import { streamObject } from 'ai';

export async function POST(req: Request) {
  const context = await req.json();

  const result = streamObject({
    model: openai('gpt-4.1'),
    output: 'enum',
    enum: ['true', 'false'],
    prompt: `Classify this statement as true or false: ${context}`,
  });

  return result.toTextStreamResponse();
}
```

Customized UI

`useObject` also provides ways to show loading and error states:

Loading State

The `isLoading` state returned by the `useObject` hook can be used for several purposes:

- To show a loading spinner while the object is generated.
- To disable the submit button.

`app/page.tsx`

```
'use client';

import { useObject } from '@ai-sdk/react';

export default function Page() {
  const { isLoading, object, submit } = useObject({
    api: '/api/notifications',
    schema: notificationSchema,
  });

  return (
    <>
      {isLoading && <Spinner />}

      <button onClick={() => submit('Messages during finals week.')} disabled={is
        Generate notifications
      </button>

      {object?.notifications?.map((notification, index) => (
        <div key={index}>
          <p>{notification?.name}</p>
          <p>{notification?.message}</p>
        </div>
      ))}
    </>
  );
}
```

Stop Handler

The `stop` function can be used to stop the object generation process. This can be useful if the user wants to cancel the request or if the server is taking too long to respond.

`app/page.tsx`

```
'use client';

import { useObject } from '@ai-sdk/react';

export default function Page() {
  const { isLoading, stop, object, submit } = useObject({
    api: '/api/notifications',
    schema: notificationSchema,
  });

  return (
    <>
```

```
      {isLoading && (
        <button type="button" onClick={() => stop()}>
          Stop
        </button>
      )}

      <button onClick={() => submit('Messages during finals week.')}>Generate not

      {object?.notifications?.map((notification, index) => (
        <div key={index}>
          <p>{notification?.name}</p>
          <p>{notification?.message}</p>
        </div>
      ))}
    </>
  );
}
```

Error State

Similarly, the `error` state reflects the error object thrown during the fetch request.
It can be used to display an error message, or to disable the submit button:

We recommend showing a generic error message to the user, such as "Something went wrong." This is a good practice to avoid leaking information from the server.

```
'use client';

import { useObject } from '@ai-sdk/react';

export default function Page() {
  const { error, object, submit } = useObject({
    api: '/api/notifications',
    schema: notificationSchema,
  });

  return (
    <>
      {error && <div>An error occurred.</div>}

      <button onClick={() => submit('Messages during finals week.')}>Generate not

      {object?.notifications?.map((notification, index) => (
        <div key={index}>
          <p>{notification?.name}</p>
          <p>{notification?.message}</p>
        </div>
      ))}
    </>
  );
}
```

Event Callbacks

`useObject` provides optional event callbacks that you can use to handle life-cycle events.

- `onFinish`: Called when the object generation is completed.
- `onError`: Called when an error occurs during the fetch request.

These callbacks can be used to trigger additional actions, such as logging, analytics, or custom UI updates.

`app/page.tsx`

```
'use client';

import { experimental_useObject as useObject } from '@ai-sdk/react';
import { notificationSchema } from './api/notifications/schema';

export default function Page() {
  const { object, submit } = useObject({
    api: '/api/notifications',
    schema: notificationSchema,
    onFinish({ object, error }) {
      // typed object, undefined if schema validation fails:
      console.log('Object generation completed:', object);

      // error, undefined if schema validation succeeds:
      console.log('Schema validation error:', error);
    },
    onError(error) {
      // error during fetch request:
      console.error('An error occurred:', error);
    },
  });

  return (
    <div>
      <button onClick={() => submit('Messages during finals week.')}>Generate not

      {object?.notifications?.map((notification, index) => (
        <div key={index}>
          <p>{notification?.name}</p>
          <p>{notification?.message}</p>
        </div>
      ))}
    </div>
  );
}
```

Configure Request Options

You can configure the API endpoint, optional headers and credentials using the `api`, `headers` and `credentials` settings.

```
const { submit, object } = useObject({
  api: '/api/use-object',

  headers: {
```

```
    'X-Custom-Header': 'CustomValue',
  },

  credentials: 'include',

  schema: yourSchema,
});
```

Streaming Custom Data

It is often useful to send additional data alongside the model's response.
For example, you may want to send status information, the message ids after storing them, or references to content that the language model is referring to.

The AI SDK provides several helpers that allows you to stream additional data to the client and attach it to the `UIMessage` parts array:

- `createUIMessageStream` : creates a data stream
- `createUIMessageStreamResponse` : creates a response object that streams data
- `pipeUIMessageStreamToResponse` : pipes a data stream to a server response object

The data is streamed as part of the response stream using Server-Sent Events.

Setting Up Type-Safe Data Streaming

First, define your custom message type with data part schemas for type safety:

```
// ai/types.ts
import { UIMessage } from 'ai';

// Define your custom message type with data part schemas
export type MyUIMessage = UIMessage<
  never, // metadata type
  {
    weather: {
      city: string;
      weather?: string;
      status: 'loading' | 'success';
    };
    notification: {
      message: string;
      level: 'info' | 'warning' | 'error';
    };
  } // data parts type
>;
```

Streaming Data from the Server

In your server-side route handler, you can create a `UIMessageStream` and then pass it to `createUIMessageStreamResponse` :

```
// route.ts
import { openai } from '@ai-sdk/openai';
import {
  createUIMessageStream,
  createUIMessageStreamResponse,
  streamText,
  convertToModelMessages,
} from 'ai';
import { MyUIMessage } from '@/ai/types';
```

```
export async function POST(req: Request) {
  const { messages } = await req.json();

  const stream = createUIMessageStream<MyUIMessage>({
    execute: ({ writer }) => {
      // 1. Send initial status (transient - won't be added to message history)
      writer.write({
        type: 'data-notification',
        data: { message: 'Processing your request...', level: 'info' },
        transient: true, // This part won't be added to message history
      });

      // 2. Send sources (useful for RAG use cases)
      writer.write({
        type: 'source',
        value: {
          type: 'source',
          sourceType: 'url',
          id: 'source-1',
          url: 'https://weather.com',
          title: 'Weather Data Source',
        },
      });

      // 3. Send data parts with loading state
      writer.write({
        type: 'data-weather',
        id: 'weather-1',
        data: { city: 'San Francisco', status: 'loading' },
      });

      const result = streamText({
        model: openai('gpt-4.1'),
        messages: convertToModelMessages(messages),
        onFinish() {
          // 4. Update the same data part (reconciliation)
          writer.write({
            type: 'data-weather',
            id: 'weather-1', // Same ID = update existing part
            data: {
              city: 'San Francisco',
              weather: 'sunny',
              status: 'success',
            },
          });

          // 5. Send completion notification (transient)
          writer.write({
            type: 'data-notification',
            data: { message: 'Request completed', level: 'info' },
            transient: true, // Won't be added to message history
          });
        },
```

```
      });

      writer.merge(result.toUIMessageStream());
    },
  });

  return createUIMessageStreamResponse({ stream });
}
```

You can also send stream data from custom backends, e.g. Python / FastAPI, using the UI Message Stream Protocol.

Types of Streamable Data

Data Parts (Persistent)

Regular data parts are added to the message history and appear in `message.parts`:

```
writer.write({
  type: 'data-weather',
  id: 'weather-1', // Optional: enables reconciliation
  data: { city: 'San Francisco', status: 'loading' },
});
```

Sources

Sources are useful for RAG implementations where you want to show which documents or URLs were referenced:

```
writer.write({
  type: 'source',
  value: {
    type: 'source',
    sourceType: 'url',
    id: 'source-1',
    url: 'https://example.com',
    title: 'Example Source',
  },
});
```

Transient Data Parts (Ephemeral)

Transient parts are sent to the client but not added to the message history. They are only accessible via the `onData` useChat handler:

```
// server
writer.write({
  type: 'data-notification',
  data: { message: 'Processing...', level: 'info' },
  transient: true, // Won't be added to message history
});

// client
const [notification, setNotification] = useState();
```

```
const { messages } = useChat({
  onData: ({ data, type }) => {
    if (type === 'data-notification') {
      setNotification({ message: data.message, level: data.level });
    }
  },
});
```

Data Part Reconciliation

When you write to a data part with the same ID, the client automatically reconciles and updates that part. This enables powerful dynamic experiences like:

- **Collaborative artifacts** - Update code, documents, or designs in real-time
- **Progressive data loading** - Show loading states that transform into final results
- **Live status updates** - Update progress bars, counters, or status indicators
- **Interactive components** - Build UI elements that evolve based on user interaction

The reconciliation happens automatically - simply use the same `id` when writing to the stream.

Processing Data on the Client

Using the onData Callback

The `onData` callback is essential for handling streaming data, especially transient parts:

```
// page.tsx
import { useChat } from '@ai-sdk/react';
import { MyUIMessage } from '@/ai/types';

const { messages } = useChat<MyUIMessage>({
  api: '/api/chat',
  onData: dataPart => {
    // Handle all data parts as they arrive (including transient parts)
    console.log('Received data part:', dataPart);

    // Handle different data part types
    if (dataPart.type === 'data-weather') {
      console.log('Weather update:', dataPart.data);
    }

    // Handle transient notifications (ONLY available here, not in message.parts)
    if (dataPart.type === 'data-notification') {
      showToast(dataPart.data.message, dataPart.data.level);
    }
  },
});
```

Important: Transient data parts are **only** available through the `onData` callback. They will not appear in the `message.parts` array since they're not added to message history.

Rendering Persistent Data Parts

You can filter and render data parts from the message parts array:

```
// page.tsx
const result = (
  <>
    {messages?.map(message => (
      <div key={message.id}>
        {/* Render weather data parts */}
        {message.parts
          .filter(part => part.type === 'data-weather')
          .map((part, index) => (
            <div key={index} className="weather-widget">
              {part.data.status === 'loading' ? (
                <>Getting weather for {part.data.city}...</>
              ) : (
                <>
                  Weather in {part.data.city}: {part.data.weather}
                </>
              )}
            </div>
          ))}

        {/* Render text content */}
        {message.parts
          .filter(part => part.type === 'text')
          .map((part, index) => (
            <div key={index}>{part.text}</div>
          ))}

        {/* Render sources */}
        {message.parts
          .filter(part => part.type === 'source')
          .map((part, index) => (
            <div key={index} className="source">
              Source: <a href={part.url}>{part.title}</a>
            </div>
          ))}
      </div>
    ))}

    <form onSubmit={handleSubmit}>
      <input
        value={input}
        onChange={e => setInput(e.target.value)}
        placeholder="Ask about the weather..."
      />
      <button type="submit">Send</button>
    </form>
  </>
);
```

Complete Example

```
'use client';

import { useChat } from '@ai-sdk/react';
import { useState } from 'react';
import { MyUIMessage } from '@/ai/types';

export default function Chat() {
  const [input, setInput] = useState('');

  const { messages, sendMessage } = useChat<MyUIMessage>({
    api: '/api/chat',
    onData: dataPart => {
      // Handle transient notifications
      if (dataPart.type === 'data-notification') {
        console.log('Notification:', dataPart.data.message);
      }
    },
  });

  const handleSubmit = (e: React.FormEvent) => {
    e.preventDefault();
    sendMessage({ text: input });
    setInput('');
  };

  return (
    <>
      {messages?.map(message => (
        <div key={message.id}>
          {message.role === 'user' ? 'User: ' : 'AI: '}

          {/* Render weather data */}
          {message.parts
            .filter(part => part.type === 'data-weather')
            .map((part, index) => (
              <span key={index} className="weather-update">
                {part.data.status === 'loading' ? (
                  <>Getting weather for {part.data.city}...</>
                ) : (
                  <>Weather in {part.data.city}: {part.data.weather}</>
                )}
              </span>
            ))}

          {/* Render text content */}
          {message.parts
            .filter(part => part.type === 'text')
            .map((part, index) => (
              <div key={index}>{part.text}</div>
            ))}
        </div>
      ))}

      <form onSubmit={handleSubmit}>
```

```
        <input
          value={input}
          onChange={e => setInput(e.target.value)}
          placeholder="Ask about the weather..."
        />
        <button type="submit">Send</button>
      </form>
    </>
  );
}
```

Use Cases

- **RAG Applications** - Stream sources and retrieved documents
- **Real-time Status** - Show loading states and progress updates
- **Collaborative Tools** - Stream live updates to shared artifacts
- **Analytics** - Send usage data without cluttering message history
- **Notifications** - Display temporary alerts and status messages

Message Metadata vs Data Parts

Both message metadata and data parts allow you to send additional information alongside messages, but they serve different purposes:

Message Metadata

Message metadata is best for **message-level information** that describes the message as a whole:

- Attached at the message level via `message.metadata`
- Sent using the `messageMetadata` callback in `toUIMessageStreamResponse`
- Ideal for: timestamps, model info, token usage, user context
- Type-safe with custom metadata types

```
// Server: Send metadata about the message
return result.toUIMessageStreamResponse({
  messageMetadata: ({ part }) => {
    if (part.type === 'finish') {
      return {
        model: part.response.modelId,
        totalTokens: part.totalUsage.totalTokens,
        createdAt: Date.now(),
      };
    }
  },
});
```

Data Parts

Data parts are best for streaming **dynamic arbitrary data**:

- Added to the message parts array via `message.parts`
- Streamed using `createUIMessageStream` and `writer.write()`
- Can be reconciled/updated using the same ID
- Support transient parts that don't persist

- Ideal for: dynamic content, loading states, interactive components

```
// Server: Stream data as part of message content
writer.write({
  type: 'data-weather',
  id: 'weather-1',
  data: { city: 'San Francisco', status: 'loading' },
});
```

For more details on message metadata, see the Message Metadata documentation.

Previous: Object Generation
Next: Error Handling

Error Handling

Error Helper Object

Each AI SDK UI hook also returns an error object that you can use to render the error in your UI. You can use the error object to show an error message, disable the submit button, or show a retry button.

We recommend showing a generic error message to the user, such as "Something went wrong." This is a good practice to avoid leaking information from the server.

```
'use client';

import { useChat } from '@ai-sdk/react';
import { useState } from 'react';

export default function Chat() {
  const [input, setInput] = useState('');
  const { messages, sendMessage, error, regenerate } = useChat();

  const handleSubmit = (e: React.FormEvent) => {
    e.preventDefault();
    sendMessage({ text: input });
    setInput('');
  };

  return (
    <div>
      {messages.map(m => (
        <div key={m.id}>
          {m.role}: {' '}
          {m.parts
            .filter(part => part.type === 'text')
            .map(part => part.text)
            .join('')}
        </div>
      ))}

      {error && (
        <>
          <div>An error occurred.</div>
          <button type="button" onClick={() => regenerate()}>
            Retry
          </button>
        </>
      )}

      <form onSubmit={handleSubmit}>
        <input
          value={input}
          onChange={e => setInput(e.target.value)}
          disabled={error != null}
```

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

Alternative: replace last message

Alternatively you can write a custom submit handler that replaces the last message when an error is present.

```
'use client';

import { useChat } from '@ai-sdk/react';
import { useState } from 'react';

export default function Chat() {
  const [input, setInput] = useState('');
  const { sendMessage, error, messages, setMessages } = useChat();

  function customSubmit(event: React.FormEvent<HTMLFormElement>) {
    event.preventDefault();

    if (error != null) {
      setMessages(messages.slice(0, -1)); // remove last message
    }

    sendMessage({ text: input });
    setInput('');
  }

  return (
    <div>
      {messages.map(m => (
        <div key={m.id}>
          {m.role}: {' '}
          {m.parts
            .filter(part => part.type === 'text')
            .map(part => part.text)
            .join('')}
        </div>
      ))}

      {error && <div>An error occurred.</div>}

      <form onSubmit={customSubmit}>
        <input
          value={input}
          onChange={e => setInput(e.target.value)}
        />
      </form>
    </div>
  );
}
```

Error Handling Callback

Errors can be processed by passing an `onError` callback function as an option to the `useChat` or `useCompletion` hooks.
The callback function receives an error object as an argument.

```
import { useChat } from '@ai-sdk/react';

export default function Page() {
  const {
    /* ... */
  } = useChat({
    // handle error:
    onError: error => {
      console.error(error);
    },
  });
}
```

Injecting Errors for Testing

You might want to create errors for testing.
You can easily do so by throwing an error in your route handler:

```
export async function POST(req: Request) {
  throw new Error('This is a test error');
}
```

Transport

The `useChat` transport system provides fine-grained control over how messages are sent to your API endpoints and how responses are processed. This is particularly useful for alternative communication protocols like WebSockets, custom authentication patterns, or specialized backend integrations.

Default Transport

By default, `useChat` uses HTTP POST requests to send messages to `/api/chat`:

```
import { useChat } from '@ai-sdk/react';

// Uses default HTTP transport
const { messages, sendMessage } = useChat();
```

This is equivalent to:

```
import { useChat } from '@ai-sdk/react';
import { DefaultChatTransport } from 'ai';

const { messages, sendMessage } = useChat({
  transport: new DefaultChatTransport({
    api: '/api/chat',
  }),
});
```

Custom Transport Configuration

Configure the default transport with custom options:

```
import { useChat } from '@ai-sdk/react';
import { DefaultChatTransport } from 'ai';

const { messages, sendMessage } = useChat({
  transport: new DefaultChatTransport({
    api: '/api/custom-chat',
    headers: {
      Authorization: 'Bearer your-token',
      'X-API-Version': '2024-01',
    },
    credentials: 'include',
  }),
});
```

Dynamic Configuration

You can also provide functions that return configuration values. This is useful for authentication tokens that need to be refreshed, or for configuration that depends on runtime conditions:

```
const { messages, sendMessage } = useChat({
  transport: new DefaultChatTransport({
    api: '/api/chat',
```

```
    headers: () => ({
      Authorization: `Bearer ${getAuthToken()}`,
      'X-User-ID': getCurrentUserId(),
    }),
    body: () => ({
      sessionId: getCurrentSessionId(),
      preferences: getUserPreferences(),
    }),
    credentials: () => 'include',
  }),
});
```

Request Transformation

Transform requests before sending to your API:

```
const { messages, sendMessage } = useChat({
  transport: new DefaultChatTransport({
    api: '/api/chat',
    prepareSendMessagesRequest: ({ id, messages, trigger, messageId }) => {
      return {
        headers: {
          'X-Session-ID': id,
        },
        body: {
          messages: messages.slice(-10), // Only send last 10 messages
          trigger,
          messageId,
        },
      };
    },
  }),
});
```

Building Custom Transports

To understand how to build your own transport, refer to the source code of the default implementation:

- **DefaultChatTransport** - The complete default HTTP transport implementation
- **HttpChatTransport** - Base HTTP transport with request handling
- **ChatTransport Interface** - The transport interface you need to implement

These implementations show you exactly how to:

- Handle the `sendMessages` method
- Process UI message streams
- Transform requests and responses
- Handle errors and connection management

The transport system gives you complete control over how your chat application communicates, enabling integration with any backend protocol or service.

Previous: Error Handling
Next: Reading UIMessage Streams

AI SDK RSC

AI SDK RSC is currently experimental. We recommend using AI SDK UI for production. For guidance on migrating from RSC to UI, see our migration guide.

- Overview
  Learn about AI SDK RSC.
- Streaming React Components
  Learn how to stream React components.
- Managing Generative UI State
  Learn how to manage generative UI state.
- Saving and Restoring States
  Learn how to save and restore states.
- Multi-step Interfaces
  Learn how to build multi-step interfaces.
- Streaming Values
  Learn how to stream values with AI SDK RSC.
- Error Handling
  Learn how to handle errors.
- Authentication
  Learn how to authenticate users.

Navigation

Previous: Stream Protocols
Next: Overview

On this page

Resources

- Docs
- Cookbook
- Providers
- Showcase
- GitHub

- Discussions

More

- Playground
- Contact Sales

About Vercel

- Next.js + Vercel
- Open Source Software
- GitHub
- X

Legal

- Privacy Policy

Advanced

This section covers advanced topics and concepts for the AI SDK and RSC API. Working with LLMs often requires a different mental model compared to traditional software development.

After these concepts, you should have a better understanding of the paradigms behind the AI SDK and RSC API, and how to use them to build more AI applications.

Previous: Migrating from RSC to UI
Next: Prompt Engineering

On this page

Resources

- Docs
- Cookbook
- Providers
- Showcase
- GitHub
- Discussions

More

- Playground
- Contact Sales

About Vercel

- Next.js + Vercel
- Open Source Software
- GitHub
- X

Legal

- Privacy Policy

Reference

API Reference

- AI SDK Core
  Switch between model providers without changing your code.
- AI SDK RSC
  Use React Server Components to stream user interfaces to the client.
- AI SDK UI
  Use hooks to integrate user interfaces that interact with language models.
- Stream Helpers
  Use special functions that help stream model generations from various providers.

Navigation

Previous: Vercel Deployment Guide
Next: AI SDK Core

On this page

Resources

- Docs
- Cookbook
- Providers
- Showcase
- GitHub
- Discussions

More

- Playground
- Contact Sales

About Vercel

- Next.js + Vercel

- Open Source Software
- GitHub
- X

Legal

- Privacy Policy

AI SDK Core

AI SDK Core is a set of functions that allow you to interact with language models and other AI models. These functions are designed to be easy-to-use and flexible, allowing you to generate text, structured data, and embeddings from language models and other AI models.

AI SDK Core contains the following main functions:

- generateText()
  Generate text and call tools from a language model.
- streamText()
  Stream text and call tools from a language model.
- generateObject()
  Generate structured data from a language model.
- streamObject()
  Stream structured data from a language model.
- embed()
  Generate an embedding for a single value using an embedding model.
- embedMany()
  Generate embeddings for several values using an embedding model (batch embedding).
- experimental_generateImage()
  Generate images based on a given prompt using an image model.
- experimental_transcribe()
  Generate a transcript from an audio file.
- experimental_generateSpeech()
  Generate speech audio from text.

It also contains the following helper functions:

- tool()
  Type inference helper function for tools.
- experimental_createMCPClient()
  Creates a client for connecting to MCP servers.
- jsonSchema()
  Creates AI SDK compatible JSON schema objects.
- zodSchema()
  Creates AI SDK compatible Zod schema objects.
- createProviderRegistry()
  Creates a registry for using models from multiple providers.
- cosineSimilarity()
  Calculates the cosine similarity between two vectors, e.g. embeddings.

- simulateReadableStream()
  Creates a ReadableStream that emits values with configurable delays.
- wrapLanguageModel()
  Wraps a language model with middleware.
- extractReasoningMiddleware()
  Extracts reasoning from the generated text and exposes it as a `reasoning` property on the result.
- simulateStreamingMiddleware()
  Simulates streaming behavior with responses from non-streaming language models.
- defaultSettingsMiddleware()
  Applies default settings to a language model.
- smoothStream()
  Smooths text streaming output.
- generateId()
  Helper function for generating unique IDs.
- createIdGenerator()
  Creates an ID generator.

Previous: Reference
Next: generateText

AI SDK UI

AI SDK UI is designed to help you build interactive chat, completion, and assistant applications with ease.
It is a framework-agnostic toolkit, streamlining the integration of advanced AI functionalities into your applications.

AI SDK UI contains the following hooks:

- useChat
  Use a hook to interact with language models in a chat interface.
- useCompletion
  Use a hook to interact with language models in a completion interface.
- useObject
  Use a hook for consuming streamed JSON objects.
- convertToModelMessages
  Convert useChat messages to ModelMessages for AI functions.
- createUIMessageStream
  Create a UI message stream to stream additional data to the client.
- createUIMessageStreamResponse
  Create a response object to stream UI messages to the client.
- pipeUIMessageStreamToResponse
  Pipe a UI message stream to a Node.js ServerResponse object.
- readUIMessageStream
  Transform a stream of UIMessageChunk objects into an AsyncIterableStream of UIMessage objects.

UI Framework Support

AI SDK UI supports the following frameworks: React, Svelte, and Vue.js.
Here is a comparison of the supported functions across these frameworks:

Function	React	Svelte	Vue.js
useChat	Chat		
useCompletion	Completion		
useObject	StructuredObject		

Contributions are welcome to implement missing features for non-React frameworks.

AI SDK RSC

AI SDK RSC is currently experimental. We recommend using AI SDK UI for production. For guidance on migrating from RSC to UI, see our migration guide.

streamUI
Use a helper function that streams React Server Components on tool execution.

createAI
Create a context provider that wraps your application and shares state between the client and language model on the server.

createStreamableUI
Create a streamable UI component that can be rendered on the server and streamed to the client.

createStreamableValue
Create a streamable value that can be rendered on the server and streamed to the client.

getAIState
Read the AI state on the server.

getMutableAIState
Read and update the AI state on the server.

useAIState
Get the AI state on the client from the context provider.

useUIState
Get the UI state on the client from the context provider.

useActions
Call server actions from the client.

Reference: Stream Helpers

Navigation

Menu

AI SDK by Vercel

Foundations

Getting Started

AI SDK Core

- Transcription
- Speech
- Language Model Middleware
- Provider & Model Management
- Error Handling
- Testing
- Telemetry

AI SDK UI

- Overview
- Chatbot
- Chatbot Message Persistence
- Chatbot Tool Usage
- Generative User Interfaces
- Completion
- Object Generation
- Streaming Custom Data
- Error Handling
- Transport
- Reading UIMessage Streams
- Message Metadata
- Stream Protocols

AI SDK RSC

- AI SDK RSC

Advanced

- Advanced

Reference

- AI SDK Core
- AI SDK UI
- AI SDK RSC
- Stream Helpers
- AI SDK Errors

Migration Guides

- Migration Guides

Troubleshooting

- Troubleshooting

Stream Helpers

Overview List

- **AIStream**
  Create a readable stream for AI responses.
- **StreamingTextResponse**
  Create a streaming response for text generations.
- **streamToResponse**
  Pipe a ReadableStream to a Node.js ServerResponse object.
- **OpenAIStream**
  Transforms the response from OpenAI's language models into a readable stream.
- **AnthropicStream**
  Transforms the response from Anthropic's language models into a readable stream.
- **AWSBedrockStream**
  Transforms the response from AWS Bedrock's language models into a readable stream.
- **AWSBedrockMessagesStream**
  Transforms the response from AWS Bedrock Message's language models into a readable stream.
- **AWSBedrockCohereStream**
  Transforms the response from AWS Bedrock Cohere's language models into a readable stream.
- **AWSBedrockLlama-2Stream**
  Transforms the response from AWS Bedrock Llama-2's language models into a readable stream.
- **CohereStream**
  Transforms the response from Cohere's language models into a readable stream.
- **GoogleGenerativeAIStream**
  Transforms the response from Google's language models into a readable stream.
- **HuggingFaceStream**
  Transforms the response from Hugging Face's language models into a readable stream.
- **LangChainStream**
  Transforms the response from LangChain's language models into a readable stream.
- **MistralStream**
  Transforms the response from Mistral's language models into a readable stream.
- **ReplicateStream**
  Transforms the response from Replicate's language models into a readable stream.
- **InkeepsStream**
  Transforms the response from Inkeeps's language models into a readable stream.

Navigation

- Previous: render (Removed)
- Next: AIStream

Talk to an expert

Resources

- Docs
- Cookbook
- Providers
- Showcase
- GitHub
- Discussions

More

- Playground
- Contact Sales

About Vercel

- Next.js + Vercel
- Open Source Software
- GitHub
- X

Legal

- Privacy Policy

AI SDK Errors

- AI_APICallError
- AI_DownloadError
- AI_EmptyResponseBodyError
- AI_InvalidArgumentError
- AI_InvalidDataContent
- AI_InvalidDataContentError
- AI_InvalidMessageRoleError
- AI_InvalidPromptError
- AI_InvalidResponseDataError
- AI_InvalidToolArgumentsError
- AI_JSONParseError
- AI_LoadAPIKeyError
- AI_LoadSettingError
- AI_MessageConversionError
- AI_NoAudioGeneratedError
- AI_NoContentGeneratedError
- AI_NoImageGeneratedError
- AI_NoTranscriptGeneratedError
- AI_NoObjectGeneratedError
- AI_NoOutputSpecifiedError
- AI_NoSuchModelError
- AI_NoSuchProviderError
- AI_NoSuchToolError
- AI_RetryError
- AI_ToolCallRepairError
- AI_ToolExecutionError
- AI_TooManyEmbeddingValuesForCallError
- AI_TypeValidationError
- AI_UnsupportedFunctionalityError

Migration Guides

- Migrate AI SDK 4.x to 5.0
- Migrate AI SDK 4.1 to 4.2
- Migrate AI SDK 4.0 to 4.1
- Migrate AI SDK 3.4 to 4.0
- Migrate AI SDK 3.3 to 3.4
- Migrate AI SDK 3.2 to 3.3
- Migrate AI SDK 3.1 to 3.2
- Migrate AI SDK 3.0 to 3.1

Versioning

- Versioning

Troubleshooting

This section is designed to help you quickly identify and resolve common issues encountered with the AI SDK, ensuring a smoother and more efficient development experience.

Report Issues

Found a bug? We'd love to hear about it in our GitHub issues.

Open GitHub Issue

Feature Requests

Want to suggest a new feature? Share it with us and the community.

Request Feature

Ask the Community

Join our GitHub discussions to browse for help and best practices.

Ask a question

Migration Guides

Check out our migration guides to help you upgrade to the latest version.

Migration Guides

Navigation

- Previous: Migrate AI SDK 3.0 to 3.1
- Next: Azure OpenAI Slow to Stream

Resources

- Docs
- Cookbook

- Providers
- Showcase
- GitHub
- Discussions

More

- Playground
- Contact Sales

About Vercel

- Next.js + Vercel
- Open Source Software
- GitHub
- X

Legal

- Privacy Policy