The OpenAI provider contains language model support for the OpenAI responses, chat, and completion APIs, as well as embedding model support for the OpenAI embeddings API.

Setup

The OpenAI provider is available in the `@ai-sdk/openai` module. You can install it with

pnpm npm yarn

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

Provider Instance

You can import the default provider instance `openai` from `@ai-sdk/openai`:

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

If you need a customized setup, you can import `createOpenAI` from `@ai-sdk/openai` and create a provider instance with your settings:

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

const openai = createOpenAI({
  // custom settings, e.g.
  headers: {
    'header-name': 'header-value',
  },
});
```

You can use the following optional settings to customize the OpenAI provider instance:

- **baseURL** *string*

  Use a different URL prefix for API calls, e.g. to use proxy servers. The default prefix is `https://api.openai.com/v1`.

- **apiKey** *string*

  API key that is being sent using the `Authorization` header. It defaults to the `OPENAI_API_KEY` environment variable.

- **name** *string*

  The provider name. You can set this when using OpenAI compatible providers to change the model provider property. Defaults to `openai`.

- **organization** *string*

  OpenAI Organization.

- **project** *string*

  OpenAI project.

- **headers** *Record<string,string>*

  Custom headers to include in the requests.

- **fetch** *(input: RequestInfo, init?: RequestInit) => Promise<Response>*

  Custom fetch ↗ implementation. Defaults to the global `fetch` function. You can use it as a middleware to intercept requests, or to provide a custom fetch implementation for e.g. testing.

Language Models

The OpenAI provider instance is a function that you can invoke to create a language model:

```
const model = openai('gpt-4.1');
```

It automatically selects the correct API based on the model id. You can also pass additional settings in the second argument:

```
const model = openai('gpt-4.1', {
  // additional settings
});
```

The available options depend on the API that's automatically chosen for the model (see below). If you want to explicitly select a specific model API, you can use `.chat` or `.completion`.

Example

You can use OpenAI language models to generate text with the `generateText` function:

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

const { text } = await generateText({
  model: openai('gpt-4.1'),
  prompt: 'Write a vegetarian lasagna recipe for 4 people.',
});
```

OpenAI language models can also be used in the `streamText`, `generateObject`, and `streamObject` functions (see AI SDK Core).

Chat Models

You can create models that call the OpenAI chat API ↗ using the `.chat()` factory method. The first argument is the model id, e.g. `gpt-4`. The OpenAI chat models support tool calls and some have multi-modal capabilities.

```
const model = openai.chat('gpt-3.5-turbo');
```

OpenAI chat models support also some model specific provider options that are not part of the standard call settings. You can pass them in the `providerOptions` argument:

```
const model = openai.chat('gpt-3.5-turbo');

await generateText({
  model,
  providerOptions: {
    openai: {
      logitBias: {
        // optional likelihood for specific tokens
        '50256': -100,
      },
      user: 'test-user', // optional unique user identifier
    },
  },
});
```

The following optional provider options are available for OpenAI chat models:

- **logitBias** *Record<number, number>*

  Modifies the likelihood of specified tokens appearing in the completion.

  Accepts a JSON object that maps tokens (specified by their token ID in the GPT tokenizer) to an associated bias value from -100 to 100. You can use this tokenizer tool to convert text to token IDs. Mathematically, the bias is added to the logits generated by the model prior to sampling. The exact effect will vary per model, but values between -1 and 1 should decrease or increase likelihood of selection; values like -100 or 100 should result in a ban or exclusive selection of the relevant token.

  As an example, you can pass `{"50256": -100}` to prevent the token from being generated.

- **logprobs** *boolean | number*

  Return the log probabilities of the tokens. Including logprobs will increase the response size and can slow down response times. However, it can be useful to better understand how the model is behaving.

Setting to true will return the log probabilities of the tokens that were generated.

Setting to a number will return the log probabilities of the top n tokens that were generated.

- **parallelToolCalls** *boolean*

  Whether to enable parallel function calling during tool use. Defaults to `true`.

- **user** *string*

  A unique identifier representing your end-user, which can help OpenAI to monitor and detect abuse. Learn more ↗.

- **reasoningEffort** *'low' | 'medium' | 'high'*

  Reasoning effort for reasoning models. Defaults to `medium`. If you use `providerOptions` to set the `reasoningEffort` option, this model setting will be ignored.

- **structuredOutputs** *boolean*

  Whether to use structured outputs. Defaults to `true`.

  When enabled, tool calls and object generation will be strict and follow the provided schema.

Reasoning

OpenAI has introduced the `o1`, `o3`, and `o4` series of reasoning models ↗. Currently, `o4-mini`, `o3`, `o3-mini`, `o1`, `o1-mini`, and `o1-preview` are available.

Reasoning models currently only generate text, have several limitations, and are only supported using `generateText` and `streamText`.

They support additional settings and response metadata:

- You can use `providerOptions` to set
  - the `reasoningEffort` option (or alternatively the `reasoningEffort` model setting), which determines the amount of reasoning the model performs.
- You can use response `providerMetadata` to access the number of reasoning tokens that the model generated.

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

const { text, usage, providerMetadata } = await generateText({
  model: openai('o3-mini'),
  prompt: 'Invent a new holiday and describe its traditions.',
  providerOptions: {
    openai: {
      reasoningEffort: 'low',
    },
  },
});

console.log(text);
console.log('Usage:', {
  ...usage,
  reasoningTokens: providerMetadata?.openai?.reasoningTokens,
});
```

System messages are automatically converted to OpenAI developer messages for reasoning models when supported. For models that do not support developer messages, such as `o1-preview`, system messages are removed and a warning is added.

Reasoning models like `o1-mini` and `o1-preview` require additional runtime inference to complete their reasoning phase before generating a response. This introduces longer latency compared to other models, with `o1-preview` exhibiting significantly more inference time than `o1-mini`.

`maxOutputTokens` is automatically mapped to `max_completion_tokens` for reasoning models.

Structured Outputs

Structured outputs are enabled by default. You can disable them by setting the `structuredOutputs` option to `false`.

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

const result = await generateObject({
  model: openai('gpt-4o-2024-08-06'),
  providerOptions: {
    openai: {
      structuredOutputs: false,
    },
  },
  schemaName: 'recipe',
  schemaDescription: 'A recipe for lasagna.',
  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.',
});

console.log(JSON.stringify(result.object, null, 2));
```

⚠ OpenAI structured outputs have several limitations, in particular around the supported schemas, and are therefore opt-in.

For example, optional schema properties are not supported. You need to change Zod `.nullish()` and `.optional()` to `.nullable()`.

Logprobs

OpenAI provides logprobs information for completion/chat models. You can access it in the `providerMetadata` object.

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

const result = await generateText({
  model: openai('gpt-4o'),
  prompt: 'Write a vegetarian lasagna recipe for 4 people.',
  providerOptions: {
    openai: {
      // this can also be a number,
      // refer to logprobs provider options section for more
      logprobs: true,
    },
  },
});

const openaiMetadata = (await result.providerMetadata)?.openai;

const logprobs = openaiMetadata?.logprobs;
```

PDF support

The OpenAI Chat API supports reading PDF files. You can pass PDF files as part of the message content using the `file` type:

```
const result = await generateText({
  model: openai('gpt-4o'),
  messages: [
    {
      role: 'user',
      content: [
        {
          type: 'text',
          text: 'What is an embedding model?',
        },
        {
          type: 'file',
          data: fs.readFileSync('./data/ai.pdf'),
          mediaType: 'application/pdf',
          filename: 'ai.pdf', // optional
        },
      ],
    },
  ],
});
```

The model will have access to the contents of the PDF file and respond to questions about it. The PDF file should be passed using the `data` field, and the `mediaType` should be set to `'application/pdf'`.

Predicted Outputs

OpenAI supports predicted outputs ↗ for `gpt-4o` and `gpt-4o-mini`. Predicted outputs help you reduce latency by allowing you to specify a base text that the model should modify. You can enable predicted outputs by adding the `prediction` option to the `providerOptions.openai` object:

```
const result = streamText({
  model: openai('gpt-4o'),
  messages: [
    {
      role: 'user',
      content: 'Replace the Username property with an Email property.',
    },
    {
      role: 'user',
      content: existingCode,
    },
  ],
  providerOptions: {
    openai: {
      prediction: {
        type: 'content',
        content: existingCode,
      },
    },
  },
});
```

OpenAI provides usage information for predicted outputs (`acceptedPredictionTokens` and `rejectedPredictionTokens`). You can access it in the `providerMetadata` object.

```
const openaiMetadata = (await result.providerMetadata)?.openai;

const acceptedPredictionTokens = openaiMetadata?.acceptedPredictionTokens;
const rejectedPredictionTokens = openaiMetadata?.rejectedPredictionTokens;
```

⚠ OpenAI Predicted Outputs have several limitations ↗, e.g. unsupported API parameters and no tool calling support.

Image Detail

You can use the `openai` provider option to set the image input detail ↗ to `high`, `low`, or `auto`:

```
const result = await generateText({
  model: openai('gpt-4o'),
  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/comic-cat.png?raw=true',

          // OpenAI specific options - image detail:
          providerOptions: {
            openai: { imageDetail: 'low' },
          },
        },
      ],
    },
  ],
});
```

⚠ Because the `UIMessage` type (used by AI SDK UI hooks like `useChat`) does not support the `providerOptions` property, you can use `convertToModelMessages` first before passing the messages to functions like `generateText` or `streamText`. For more details on `providerOptions` usage, see here.

Distillation

OpenAI supports model distillation for some models. If you want to store a generation for use in the distillation process, you can add the `store` option to the `providerOptions.openai` object. This will save the generation to the OpenAI platform for later use in distillation.

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

async function main() {
  const { text, usage } = await generateText({
    model: openai('gpt-4o-mini'),
    prompt: 'Who worked on the original macintosh?',
    providerOptions: {
      openai: {
        store: true,
        metadata: {
          custom: 'value',
        },
      },
    },
  });

  console.log(text);
  console.log();
  console.log('Usage:', usage);
}

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

Prompt Caching

OpenAI has introduced Prompt Caching for supported models including `gpt-4o`, `gpt-4o-mini`, `o1-preview`, and `o1-mini`.

- Prompt caching is automatically enabled for these models, when the prompt is 1024 tokens or longer. It does not need to be explicitly enabled.
- You can use response `providerMetadata` to access the number of prompt tokens that were a cache hit.
- Note that caching behavior is dependent on load on OpenAI's infrastructure. Prompt prefixes generally remain in the cache following 5-10 minutes of inactivity before they are evicted, but during off-peak periods they may persist for up to an hour.

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

const { text, usage, providerMetadata } = await generateText({
  model: openai('gpt-4o-mini'),
  prompt: `A 1024-token or longer prompt...`,
});

console.log(`usage:`, {
  ...usage,
  cachedPromptTokens: providerMetadata?.openai?.cachedPromptTokens,
});
```

Audio Input

With the `gpt-4o-audio-preview` model, you can pass audio files to the model.

⚠ The `gpt-4o-audio-preview` model is currently in preview and requires at least some audio inputs. It will not work with non-audio data.

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

Responses Models

You can use the OpenAI responses API with the `openai.responses(modelId)` factory method.

```
const model = openai.responses('gpt-4o-mini');
```

Further configuration can be done using OpenAI provider options. You can validate the provider options using the `OpenAIResponsesProviderOptions` type.

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

const result = await generateText({
  model: openai.responses('gpt-4o-mini'),
  providerOptions: {
    openai: {
      parallelToolCalls: false,
      store: false,
      user: 'user_123',
      // ...
    } satisfies OpenAIResponsesProviderOptions,
  },
  // ...
});
```

The following provider options are available:

- **parallelToolCalls** *boolean* Whether to use parallel tool calls. Defaults to `true`.
- **store** *boolean* Whether to store the generation. Defaults to `true`.
- **metadata** *Record<string, string>* Additional metadata to store with the generation.
- **previousResponseId** *string* The ID of the previous response. You can use it to continue a conversation. Defaults to `undefined`.
- **instructions** *string* Instructions for the model. They can be used to change the system or developer message when continuing a conversation using the `previousResponseId` option. Defaults to `undefined`.
- **user** *string* A unique identifier representing your end-user, which can help OpenAI to monitor and detect abuse. Defaults to `undefined`.
- **reasoningEffort** *'low' | 'medium' | 'high'* Reasoning effort for reasoning models. Defaults to `medium`. If you use `providerOptions` to set the `reasoningEffort` option, this model setting will be ignored.
- **reasoningSummary** *'auto' | 'detailed'* Controls whether the model returns its reasoning process. Set to `'auto'` for a condensed summary, `'detailed'` for more comprehensive reasoning. Defaults to `undefined` (no reasoning summaries). When enabled, reasoning summaries appear in the stream as events with type `'reasoning'` and in non-streaming responses within the `reasoning` field.
- **strictSchemas** *boolean* Whether to use strict JSON schemas in tools and when generating JSON outputs. Defaults to `true`.
- **parallelToolCalls** *boolean* Whether to enable parallel function calling during tool use. Default to true.

The OpenAI responses provider also returns provider-specific metadata:

```
const { providerMetadata } = await generateText({
  model: openai.responses('gpt-4o-mini'),
});

const openaiMetadata = providerMetadata?.openai;
```

The following OpenAI-specific metadata is returned:

- **responseId** *string* The ID of the response. Can be used to continue a conversation.
- **cachedPromptTokens** *number* The number of prompt tokens that were a cache hit.
- **reasoningTokens** *number* The number of reasoning tokens that the model generated.

Web Search

The OpenAI responses provider supports web search through the `openai.tools.webSearchPreview` tool.

You can force the use of the web search tool by setting the `toolChoice` parameter to `{ type: 'tool', toolName: 'web_search_preview' }`.

```
const result = await generateText({
  model: openai.responses('gpt-4o-mini'),
  prompt: 'What happened in San Francisco last week?',
  tools: {
    web_search_preview: openai.tools.webSearchPreview({
      // optional configuration:
      searchContextSize: 'high',
      userLocation: {
        type: 'approximate',
        city: 'San Francisco',
        region: 'California',
      },
    }),
  },
  // Force web search tool:
  toolChoice: { type: 'tool', toolName: 'web_search_preview' },
});

// URL sources
const sources = result.sources;
```

Reasoning Summaries

For reasoning models like `o3-mini`, `o3`, and `o4-mini`, you can enable reasoning summaries to see the model's thought process. Different models support different summarizers—for example, `o4-mini` supports detailed summaries. Set `reasoningSummary: "auto"` to automatically receive the richest level available.

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

const result = streamText({
  model: openai.responses('o4-mini'),
  prompt: 'Tell me about the Mission burrito debate in San Francisco.',
  providerOptions: {
    openai: {
      reasoningSummary: 'detailed', // 'auto' for condensed or 'detailed' for comprehensive
    },
  },
});

for await (const part of result.fullStream) {
  if (part.type === 'reasoning') {
    console.log(`Reasoning: ${part.textDelta}`);
  } else if (part.type === 'text-delta') {
    process.stdout.write(part.textDelta);
  }
}
```

For non-streaming calls with `generateText`, the reasoning summaries are available in the `reasoning` field of the response:

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

const result = await generateText({
  model: openai.responses('o3-mini'),
  prompt: 'Tell me about the Mission burrito debate in San Francisco.',
  providerOptions: {
    openai: {
      reasoningSummary: 'auto',
    },
  },
});
console.log('Reasoning:', result.reasoning);
```

Learn more about reasoning summaries in the OpenAI documentation ↗.

PDF support

The OpenAI Responses API supports reading PDF files. You can pass PDF files as part of the message content using the `file` type:

```
const result = await generateText({
  model: openai.responses('gpt-4o'),
  messages: [
    {
      role: 'user',
      content: [
        {
          type: 'text',
          text: 'What is an embedding model?',
        },
        {
          type: 'file',
          data: fs.readFileSync('./data/ai.pdf'),
          mediaType: 'application/pdf',
          filename: 'ai.pdf', // optional
        },
      ],
    },
  ],
});
```

The model will have access to the contents of the PDF file and respond to questions about it. The PDF file should be passed using the `data` field, and the `mediaType` should be set to `'application/pdf'`.

Structured Outputs

The OpenAI Responses API supports structured outputs. You can enforce structured outputs using `generateObject` or `streamObject`, which expose a `schema` option. Additionally, you can pass a Zod or JSON Schema object to the `experimental_output` option when using `generateText` or `streamText`.

```
// Using generateObject
const result = await generateObject({
  model: openai.responses('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.',
});

// Using generateText
const result = await generateText({
  model: openai.responses('gpt-4.1'),
  prompt: 'How do I make a pizza?',
  experimental_output: Output.object({
    schema: z.object({
      ingredients: z.array(z.string()),
      steps: z.array(z.string()),
    }),
  }),
});
```

Completion Models

You can create models that call the OpenAI completions API using the `.completion()` factory method. The first argument is the model id. Currently only `gpt-3.5-turbo-instruct` is supported.

```
const model = openai.completion('gpt-3.5-turbo-instruct');
```

OpenAI completion models support also some model specific settings that are not part of the standard call settings. You can pass them as an options argument:

```
const model = openai.completion('gpt-3.5-turbo-instruct');

await model.doGenerate({
  providerOptions: {
    openai: {
      echo: true, // optional, echo the prompt in addition to the completion
      logitBias: {
        // optional likelihood for specific tokens
        '50256': -100,
      },
      suffix: 'some text', // optional suffix that comes after a completion of inserted text
      user: 'test-user', // optional unique user identifier
    },
  },
});
```

The following optional provider options are available for OpenAI completion models:

- **echo**: *boolean*

  Echo back the prompt in addition to the completion.

- **logitBias** *Record<number, number>*

  Modifies the likelihood of specified tokens appearing in the completion.

  Accepts a JSON object that maps tokens (specified by their token ID in the GPT tokenizer) to an associated bias value from -100 to 100. You can use this tokenizer tool to convert text to token IDs. Mathematically, the bias is added to the logits generated by the model prior to sampling. The exact effect will vary per model, but values between -1 and 1 should decrease or increase likelihood of selection; values like -100 or 100 should result in a ban or exclusive selection of the relevant token.

  As an example, you can pass `{"50256": -100}` to prevent the <|endoftext|> token from being generated.

- **logprobs** *boolean | number*

  Return the log probabilities of the tokens. Including logprobs will increase the response size and can slow down response times. However, it can be useful to better understand how the model is behaving.

  Setting to true will return the log probabilities of the tokens that were generated.

  Setting to a number will return the log probabilities of the top n tokens that were generated.

- **suffix** *string*

  The suffix that comes after a completion of inserted text.

- **user** *string*

  A unique identifier representing your end-user, which can help OpenAI to monitor and detect abuse. Learn more ↗ .