OpenAI Provider

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 bun

```
$ 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';
```


penAI from `@ai-sdk/openai` and

```
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>*

  al `fetch` function. You can use it as
  ustom fetch implementation for e.g.

Language Models

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

```
const model = openai('gpt-5');
```

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-5', {
  // 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-5'),
  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).

ing the `.chat()` factory method.

AI chat models support tool calls

and some have multi-modal capabilities.

```
const model = openai.chat('gpt-5');
```

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-5');

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

  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.

ogprobs will increase the response

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** *'minimal' | '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.

- **maxCompletionTokens** *number*

  Maximum number of completion tokens to generate. Useful for reasoning models.

- **store** *boolean*

  Whether to enable persistence in Responses API.

- **metadata** *Record<string, string>*

  Metadata to associate with the request.

Service tier for the request. Set to 'flex' for 50% cheaper processing at the cost of increased latency. Only available for o3, o4-mini, and gpt-5 models. Defaults to 'auto'.

- **strictJsonSchema** *boolean*

  Whether to use strict JSON schema validation. Defaults to `false`.

- **textVerbosity** *'low' | 'medium' | 'high'*

  Controls the verbosity of the model's responses. Lower values will result in more concise responses, while higher values will result in more verbose responses.

- **promptCacheKey** *string*

  A cache key for manual prompt caching control. Used by OpenAI to cache responses for similar requests to optimize your cache hit rates.

- **safetyIdentifier** *string*

  A stable identifier used to help detect users of your application that may be violating OpenAI's usage policies. The IDs should be a string that uniquely identifies each user.

Reasoning

OpenAI has introduced the `o1`, `o3`, and `o4` series of reasoning models. Currently, `o4-mini`, `o3`, `o3-mini`, and `o1` are available via both the chat and responses APIs. The models `codex-mini-latest` and `computer-use-preview` are available only via the responses API.

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

```

const { text, usage, providerMetadata } = await generateText({
  model: openai('gpt-5'),
  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.

Reasoning models require additional runtime inference to complete their reasoning phase before generating a response. This introduces longer latency compared to other models.

`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({
```



```
  },
```

```
12    schemaName: 'recipe',
13    schemaDescription: 'A recipe for lasagna.',
14    schema: z.object({
15      name: z.string(),
16      ingredients: z.array(
17        z.object({
18          name: z.string(),
19          amount: z.string(),
20        }),
21      ),
22      steps: z.array(z.string()),
23    }),
24    prompt: 'Generate a lasagna recipe.',
25  });
26
27  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-5'),
  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;
```

Image Support

The OpenAI Chat API supports Image inputs for appropriate models. You can pass Image files as part of the message content using the 'image' type:

```
const result = await generateText({
  model: openai('gpt-5'),
  messages: [
    {
      role: 'user',
      content: [
        {
          type: 'text',
          text: 'Please describe the image.',
        },
        {
          type: 'image',
          image: fs.readFileSync('./data/image.png'),
        },
      ],
    },
  ],
});
```


d to questions about it. The image

You can also pass the URL of an image.

```
{
  type: 'image',
  image: 'https://sample.edu/image.png',
}
```

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-5'),
  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'`.

You can also pass a file-id from the OpenAI Files API.

```
{
  type: 'file',
  data: 'file-8EFBcWHsQxZV7YGezBC1fq',
  mediaType: 'application/pdf',
}
```

You can also pass the URL of a PDF.

```
{
  type: 'file',
  data: 'https://sample.edu/example.pdf',
  mediaType: 'application/pdf',
  filename: 'ai.pdf', // optional
}
```

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-5'),
  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-5'),
  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-c

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


> ⚠ e `useChat`) does not support the
> Messages 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` and `gpt-4o-mini`.

odels, when the prompt is 1024

bled.

the number of prompt tokens that

- 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,
});
```

To improve cache hit rates, you can manually control caching using the `promptCacheKey` option:

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

const { text, usage, providerMetadata } = await generateText({
  model: openai('gpt-5'),
  prompt: `A 1024-token or longer prompt...`,
  providerOptions: {
    openai: {
      promptCacheKey: 'my-custom-cache-key-123',
    },
  },
});

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

Audio Input

dio 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-5');
```

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';
```

```
      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`.

  When using reasoning models (o1, o3, o4-mini) with multi-step tool calls and `store: false`, include `['reasoning.encrypted_content']` in the `include` option to ensure reasoning content is available across conversation steps.
- **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** *'minimal' | '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
  o reasoning summaries). When
  as events with type `'reasoning'`
  ng field.

- **strictJsonSchema** *boolean* Whether to use strict JSON schema validation. Defaults to `false`.
- **serviceTier** *'auto' | 'flex' | 'priority'* Service tier for the request. Set to 'flex' for 50% cheaper processing at the cost of increased latency (available for o3, o4-mini, and gpt-5 models). Set to 'priority' for faster processing with Enterprise access (available for gpt-4, gpt-5, gpt-5-mini, o3, o4-mini; gpt-5-nano is not supported). Defaults to 'auto'.
- **textVerbosity** *'low' | 'medium' | 'high'* Controls the verbosity of the model's response. Lower values result in more concise responses, while higher values result in more verbose responses. Defaults to `'medium'`.
- **include** *Array<string>* Specifies additional content to include in the response. Supported values: `['reasoning.encrypted_content']` for accessing reasoning content across conversation steps, and `['file_search_call.results']` for including file search results in responses. Defaults to `undefined`.
- **promptCacheKey** *string* A cache key for manual prompt caching control. Used by OpenAI to cache responses for similar requests to optimize your cache hit rates.
- **safetyIdentifier** _string_0 A stable identifier used to help detect users of your application that may be violating OpenAI's usage policies. The IDs should be a string that uniquely identifies each user.

The OpenAI responses provider also returns provider-specific metadata:

```
const { providerMetadata } = await generateText({
  model: openai.responses('gpt-5'),
});

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.

ot tokens that were a cache hit.

tokens that the model generated.

Web Search

The OpenAI responses API 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' }`.

```
1  const result = await generateText({
2    model: openai.responses('gpt-5'),
3    prompt: 'What happened in San Francisco last week?',
4    tools: {
5      web_search_preview: openai.tools.webSearchPreview({
6        // optional configuration:
7        searchContextSize: 'high',
8        userLocation: {
9          type: 'approximate',
10         city: 'San Francisco',
11         region: 'California',
12       },
13     }),
14   },
15   // Force web search tool:
16   toolChoice: { type: 'tool', toolName: 'web_search_preview' },
17 });
18
19 // URL sources
20 const sources = result.sources;
```

Reasoning Output

For reasoning models like `gpt-5`, 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.

```
1  import { openai } from '@ai-sdk/openai';
2  import { streamText } from 'ai';
3
4  const result = streamText({
```



```
ebate in San Francisco.',

for condensed or 'detailed' for com
```

```
  },
});

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('gpt-5'),
  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 ↗.

Verbosity Control

You can control the length and detail of model responses using the `textVerbosity` parameter:

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

const result = await generateText({
```



```
rst pet dog.',

se, 'medium' (default), or 'high' f
    },
```

```
  },
});
```

The `textVerbosity` parameter scales output length without changing the underlying prompt:

- `'low'`: Produces terse, minimal responses
- `'medium'`: Balanced detail (default)
- `'high'`: Verbose responses with comprehensive detail

File Search

The OpenAI responses API supports file search through the `openai.tools.fileSearch` tool.

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

```
const result = await generateText({
  model: openai.responses('gpt-5'),
  prompt: 'What does the document say about user authentication?',
  tools: {
    file_search: openai.tools.fileSearch({
      // optional configuration:
      vectorStoreIds: ['vs_123', 'vs_456'],
      maxNumResults: 10,
      ranking: {
        ranker: 'auto',
      },
      filters: {
        type: 'and',
        filters: [
          { key: 'author', type: 'eq', value: 'John Doe' },
          { key: 'date', type: 'gte', value: '2023-01-01' },
        ],
      },
    }),
  },
  // Force file search tool:
  toolChoice: { type: 'tool', toolName: 'file_search' },
```

The tool must be named `file_search` when using OpenAI's file search functionality. This name is required by OpenAI's API specification and cannot be customized.

Code Interpreter

The OpenAI responses API supports the code interpreter tool through the `openai.tools.codeInterpreter` tool. This allows models to write and execute Python code.

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

const result = await generateText({
  model: openai.responses('gpt-5'),
  prompt: 'Write and run Python code to calculate the factorial of 10',
  tools: {
    code_interpreter: openai.tools.codeInterpreter({
      // optional configuration:
      container: {
        fileIds: ['file-123', 'file-456'], // optional file IDs to make available
      },
    }),
  },
});
```

The code interpreter tool can be configured with:

- **container**: Either a container ID string or an object with `fileIds` to specify uploaded files that should be available to the code interpreter

The tool must be named `code_interpreter` when using OpenAI's code interpreter functionality. This name is required by OpenAI's API specification and cannot be customized.

Image Support

opropriate models. You can pass

ıge' type:

```
  model: openai.responses('gpt-5'),
  messages: [
    {
      role: 'user',
      content: [
        {
          type: 'text',
          text: 'Please describe the image.',
        },
        {
          type: 'image',
          image: fs.readFileSync('./data/image.png'),
        },
      ],
    },
  ],
});
```

The model will have access to the image and will respond to questions about it. The image should be passed using the `image` field.

You can also pass a file-id from the OpenAI Files API.

```
{
  type: 'image',
  image: 'file-8EFBcWHsQxZV7YGezBC1fq'
}
```

You can also pass the URL of an image.

```
{
  type: 'image',
  image: 'https://sample.edu/image.png',
}
```

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:

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

You can also pass a file-id from the OpenAI Files API.

```
{
  type: 'file',
  data: 'file-8EFBcWHsQxZV7YGezBC1fq',
  mediaType: 'application/pdf',
}
```

You can also pass the URL of a pdf.

```
{
  type: 'file',
  data: 'https://sample.edu/example.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'`.

You can enforce structured outputs 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');
```


ecific settings that are not part of

ns 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 in
      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

can be useful to better understand

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 ↗.

Model Capabilities

Model	Image Input	Audio Input	Object Generation	Tool Usage
`gpt-4.1`	✓	×	✓	✓
`gpt-4.1-mini`	✓	×	✓	✓
`gpt-4.1-nano`	✓	×	✓	✓
`gpt-4o`	✓	×	✓	✓
`gpt-4o-mini`	✓	×	✓	✓
`gpt-4o-audio-preview`	×	✓	✓	✓
`gpt-4-turbo`	✓	×	✓	✓
`gpt-4`	×	×	✓	✓
`gpt-3.5-turbo`	×	×	✓	✓
`o1`	✓	×	✓	✓
`o3-mini`	×	×	✓	✓
`o3`	✓	×	✓	✓
`o4-mini`	✓	×	✓	✓
			×	×
			✓	✓

Model	Image Input	Audio Input	Object Generation	Tool Usage
`gpt-5-mini`	✓	✗	✓	✓
`gpt-5-nano`	✓	✗	✓	✓
`gpt-5-chat-latest`	✓	✗	✗	✗

> The table above lists popular models. Please see the OpenAI docs ↗ for a full list of available models. The table above lists popular models. You can also pass any available provider model ID as a string if needed.

Embedding Models

You can create models that call the OpenAI embeddings API ↗ using the `.textEmbedding()` factory method.

```
const model = openai.textEmbedding('text-embedding-3-large');
```

OpenAI embedding models support several additional provider options. You can pass them as an options argument:

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

const { embedding } = await embed({
  model: openai.textEmbedding('text-embedding-3-large'),
  value: 'sunny day at the beach',
  providerOptions: {
    openai: {
      dimensions: 512, // optional, number of dimensions for the embedding
      user: 'test-user', // optional unique user identifier
```


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

- **dimensions**: *number*

  The number of dimensions the resulting output embeddings should have. Only supported in text-embedding-3 and later models.

- **user** *string*

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

Model Capabilities

Model	Default Dimensions	Custom Dimensions
`text-embedding-3-large`	3072	✓
`text-embedding-3-small`	1536	✓
`text-embedding-ada-002`	1536	×

Image Models

You can create models that call the OpenAI image generation API ↗ using the `.image()` factory method.

```
const model = openai.image('dall-e-3');
```

ⓘ Dall-E models do not support the `aspectRatio` parameter. Use the `size` parameter instead.

Model	Sizes
`gpt-image-1`	1024x1024, 1536x1024, 1024x1536
`dall-e-3`	1024x1024, 1792x1024, 1024x1792
`dall-e-2`	256x256, 512x512, 1024x1024

You can pass optional `providerOptions` to the image model. These are prone to change by OpenAI and are model dependent. For example, the `gpt-image-1` model supports the `quality` option:

```
const { image, providerMetadata } = await generateImage({
  model: openai.image('gpt-image-1'),
  prompt: 'A salamander at sunrise in a forest pond in the Seychelles.',
  providerOptions: {
    openai: { quality: 'high' },
  },
});
```

For more on `generateImage()` see Image Generation.

OpenAI's image models may return a revised prompt for each image. It can be access at `providerMetadata.openai.images[0]?.revisedPrompt`.

For more information on the available OpenAI image model options, see the OpenAI API reference ↗.

Transcription Models

You can create models that call the OpenAI transcription API ↗ using the `.transcription()` factory method.

```
const model = openai.transcription('whisper-1');
```

You can also pass additional provider-specific options using the `providerOptions` argument. For example, supplying the input language in ISO-639-1 (e.g. `en`) format will improve accuracy and latency.

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

const result = await transcribe({
  model: openai.transcription('whisper-1'),
  audio: new Uint8Array([1, 2, 3, 4]),
  providerOptions: { openai: { language: 'en' } },
});
```

To get word-level timestamps, specify the granularity:

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

const result = await transcribe({
  model: openai.transcription('whisper-1'),
  audio: new Uint8Array([1, 2, 3, 4]),
  providerOptions: {
    openai: {
      //timestampGranularities: ['word'],
      timestampGranularities: ['segment'],
    },
  },
});

// Access word-level timestamps
console.log(result.segments); // Array of segments with startSecond/endSecond
```

The following provider options are available:

- **timestampGranularities** *string[]* The granularity of the timestamps in the transcription.

d'], ['segment'], and ['word',
segment timestamps, but
y.

- **language** *string* The language of the input audio. Supplying the input language in ISO-639-1 format (e.g. 'en') will improve accuracy and latency. Optional.
- **prompt** *string* An optional text to guide the model's style or continue a previous audio segment. The prompt should match the audio language. Optional.
- **temperature** *number* The sampling temperature, between 0 and 1. Higher values like 0.8 will make the output more random, while lower values like 0.2 will make it more focused and deterministic. If set to 0, the model will use log probability to automatically increase the temperature until certain thresholds are hit. Defaults to 0. Optional.
- **include** *string[]* Additional information to include in the transcription response.

Model Capabilities

Model	Transcription	Duration	Segments	Language
`whisper-1`	✓	✓	✓	✓
`gpt-4o-mini-transcribe`	✓	×	×	×
`gpt-4o-transcribe`	✓	×	×	×

Speech Models

You can create models that call the OpenAI speech API using the `.speech()` factory method.

The first argument is the model id e.g. `tts-1`.

```
const model = openai.speech('tts-1');
```


ing the `providerOptions`
enerated audio.

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

const result = await generateSpeech({
  model: openai.speech('tts-1'),
  text: 'Hello, world!',
  providerOptions: { openai: {} },
});
```

- **instructions** *string* Control the voice of your generated audio with additional instructions e.g. "Speak in a slow and steady tone". Does not work with `tts-1` or `tts-1-hd`. Optional.
- **response_format** *string* The format to audio in. Supported formats are `mp3`, `opus`, `aac`, `flac`, `wav`, and `pcm`. Defaults to `mp3`. Optional.
- **speed** *number* The speed of the generated audio. Select a value from 0.25 to 4.0. Defaults to 1.0. Optional.

Model Capabilities

Model	Instructions
`tts-1`	✓
`tts-1-hd`	✓
`gpt-4o-mini-tts`	✓

Previous
Vercel

Next
Azure OpenAI