

# 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.

```
1  const result = await generateText({
2    model: yourModel,
3    prompt: 'Invent a new holiday and describe its tra
4  });
```


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

```
1  const result = await generateText({
2    model: yourModel,
3    prompt:
4      `I am planning a trip to ${destination} for ${length}
5      `Please suggest the best tourist activities for
6  });
```

## 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.

```
1  const result = await generateText({
2    model: yourModel,
3    system:
4      `You help planning travel itineraries. ` +
5      `Respond to the users' request with a list ` +
6      `of the best stops to make in their destination.
7    prompt:
8      `I am planning a trip to ${destination} for ${length}
9      `Please suggest the best tourist activities for
10 });
```

 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.

```
1  const result = await streamUI({
2    model: yourModel,
3    messages: [
4      { role: 'user', content: 'Hi!' },
5      { role: 'assistant', content: 'Hello, how can I help you?' },
6      { role: 'user', content: 'Where can I buy the best Currywurst?' },
7    ],
8  });
```

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.](#)

## 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.

```
1  const result = await generateText({
2    model: yourModel,
3    messages: [
4      {
5        role: 'user',
6        content: [
7          {
8            type: 'text',
9            text: 'Where can I buy the best Currywurst?'
10           },
11        ],
12      },
13    ],
14  });
```

```

11     ],
12     },
13   ],
14 });

```

## 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. `data:image/png;base64, ...`
- 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)

```

1  const result = await generateText({
2    model,
3    messages: [
4      {
5        role: 'user',
6        content: [
7          { type: 'text', text: 'Describe the image in'
8          {
9            type: 'image',
10           image: fs.readFileSync('./data/comic-cat.p
11         },
12       ],
13     },
14   ],
15 });

```

## Example: Base-64 encoded image (string)

```

1  const result = await generateText({
2    model: yourModel,
3    messages: [
4      {
5        role: 'user',
6        content: [
7          { type: 'text', text: 'Describe the image in'
8          {
9            type: 'image',
10           image: fs.readFileSync('./data/comic-cat.p
11         },
12       ],
13     },
14   ],
15 });

```

### Example: Image URL (string)

```

1  const result = await generateText({
2    model: yourModel,
3    messages: [
4      {
5        role: 'user',
6        content: [
7          { type: 'text', text: 'Describe the image in
8          {
9            type: 'image',
10           image:
11             'https://github.com/vercel/ai/blob/main/
12         },
13       ],
14     },
15   ],
16 });

```

### 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. `data:image/png;base64, ...`
- 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

```

1  import { google } from '@ai-sdk/google';
2  import { generateText } from 'ai';
3
4  const result = await generateText({
5    model: google('gemini-1.5-flash'),
6    messages: [
7      {
8        role: 'user',
9        content: [
10         { type: 'text', text: 'What is the file about?' },
11         {
12           type: 'file',
13           mimeType: 'application/pdf',
14           data: fs.readFileSync('./data/example.pdf'),
15           filename: 'example.pdf', // optional, not required
16         },
17       ],
18     },
19   ],
20 });

```

### Example: mp3 audio file from Buffer

```

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

```

```

5    model: openai('gpt-4o-audio-preview'),
6    messages: [
7      {
8        role: 'user',
9        content: [
10         { type: 'text', text: 'What is the audio say
11         {
12           type: 'file',
13           mimeType: 'audio/mpeg',
14           data: fs.readFileSync('./data/galileo.mp3')
15         },
16       ],
17     },
18   ],
19 });

```

## 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

```

1  const result = await generateText({
2    model: yourModel,
3    messages: [
4      { role: 'user', content: 'Hi!' },
5      { role: 'assistant', content: 'Hello, how can I
6    ],
7  });

```

### Example: Assistant message with text content in array

```

1  const result = await generateText({
2    model: yourModel,
3    messages: [
4      { role: 'user', content: 'Hi!' },
5      {
6        role: 'assistant',
7        content: [{ type: 'text', text: 'Hello, how ca
8      },
9    ],
10  });

```


### Example: Assistant message with tool call content

```

1  const result = await generateText({
2    model: yourModel,
3    messages: [
4      { role: 'user', content: 'How many calories are
5      {
6        role: 'assistant',
7        content: [
8          {
9            type: 'tool-call',
10           toolCallId: '12345',
11           toolName: 'get-nutrition-data',
12           args: { cheese: 'Roquefort' },
13         },
14       ],
15     },
16   ],
17 });

```

## 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.

```

1  const result = await generateText({
2    model: yourModel,
3    messages: [
4      { role: 'user', content: 'Generate an image of a
5      {
6        role: 'assistant',
7        content: [
8          {
9            type: 'file',
10           mimeType: 'image/png',
11           data: fs.readFileSync('./data/roquefort.jpg'),
12         },
13       ],
14     },
15   ],
16 });

```

## 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 result parts. A single assistant message can call multiple tools, and a single tool message can contain multiple tool results.

```
1  const result = await generateText({
2    model: yourModel,
3    messages: [
4      {
5        role: 'user',
6        content: [
7          {
8            type: 'text',
9            text: 'How many calories are in this block',
10           },
11           { type: 'image', image: fs.readFileSync('./c
12         ],
13       },
14       {
15         role: 'assistant',
16         content: [
17           {
18             type: 'tool-call',
19             toolCallId: '12345',
20             toolName: 'get-nutrition-data',
21             args: { cheese: 'Roquefort' },
22           },
23           // there could be more tool calls here (para
24         ],
25       },
26       {
27         role: 'tool',
28         content: [
29           {
30             type: 'tool-result',
31             toolCallId: '12345', // needs to match the
32             toolName: 'get-nutrition-data',
33             result: {
34               name: 'Cheese, roquefort',
35               calories: 369,
36               fat: 31,
37               protein: 22,
38             },
39           },
40           // there could be more tool results here (pa
41         ],
42       },
```

```
43     ],  
44     });
```

## 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.

```

1  const result = await generateText({
2    model: yourModel,
3    messages: [
4      // ...
5      {
6        role: 'tool',
7        content: [
8          {
9            type: 'tool-result',
10           toolCallId: '12345', // needs to match the
11           toolName: 'get-nutrition-data',
12           // for models that do not support multi-part
13           // you can include a regular result part:
14           result: {
15             name: 'Cheese, roquefort',
16             calories: 369,
17             fat: 31,
18             protein: 22,
19           },
20           // for models that support multi-part tool
21           // you can include a multi-part content part
22           content: [
23             {
24               type: 'text',
25               text: 'Here is an image of the nutrition',
26             },
27             {
28               type: 'image',
29               data: fs.readFileSync('./data/roquefort.jpg'),
30               mimeType: 'image/png',
31             },
32           ],
33         },
34       ],
35     },
36   ],
37 });

```

## 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.

```

1  const result = await generateText({
2    model: yourModel,
3    messages: [
4      { role: 'system', content: 'You help planning to' },
5      {
6        role: 'user',

```

```
7         content:
8             'I am planning a trip to Berlin for 3 days.'
9         },
10    ],
11 });
```