LangChain Docs

Core components

Messages

Copy page

Messages are the fundamental unit of context for models in LangChain. They represent the input and output of models, carrying both the content and metadata needed to represent the state of a conversation when interacting with an LLM.

Messages are objects that contain:

- **Role** - Identifies the message type (e.g. `system` , `user`)
- **Content** - Represents the actual content of the message (like text, images, audio, documents, etc.)
- **Metadata** - Optional fields such as response information, message IDs, and token usage

LangChain provides a standard message type that works across all model providers, ensuring consistent behavior regardless of the model being called.

Basic usage

The simplest way to use messages is to create message objects and pass them to a model when **invoking**.

```
const systemMsg = new SystemMessage("You are a helpful assistant.");
const humanMsg = new HumanMessage("Hello, how are you?");

const messages = [systemMsg, humanMsg];
const response = await model.invoke(messages);  // Returns AIMessage
```

Text prompts

Text prompts are strings - ideal for straightforward generation tasks where you don't need to retain conversation history.

```
const response = await model.invoke("Write a haiku about spring");
```

Use text prompts when:

- You have a single, standalone request
- You don't need conversation history
- You want minimal code complexity

Message prompts

Alternatively, you can pass in a list of messages to the model by providing a list of message objects.

LangChain Docs

LangChain Docs

Core components › **Messages**

LangChain Docs

LangChain Docs

Core components › **Messages**

LangChain Docs

LangChain Docs

LangChain Docs

LangChain Docs

LangChain Docs

Core components › **Messages**

LangChain Docs

LangChain Docs

Core components › **Messages**

LangChain Docs

Core components › **Messages**

LangChain Docs

LangChain Docs

Core components › **Messages**

LangChain Docs

LangChain Docs

LangChain Docs

Core components › **Messages**

LangChain Docs

Core components › **Messages**

LangChain Docs

Core components › **Messages**

LangChain Docs

LangChain Docs

LangChain Docs

Core components › **Messages**

LangChain Docs

Core components › **Messages**

```
import { SystemMessage, HumanMessage, AIMessage } from "langchain";

const messages = [
  new SystemMessage("You are a poetry expert"),
  new HumanMessage("Write a haiku about spring"),
  new AIMessage("Cherry blossoms bloom..."),
];
const response = await model.invoke(messages);
```

Use message prompts when:

Managing multi-turn conversations

Working with multimodal content (images, audio, files)

Including system instructions

Dictionary format

You can also specify messages directly in OpenAI chat completions format.

```
const messages = [
  { role: "system", content: "You are a poetry expert" },
  { role: "user", content: "Write a haiku about spring" },
  { role: "assistant", content: "Cherry blossoms bloom..." },
];
const response = await model.invoke(messages);
```

ssage types

AI message - Responses generated by the model, including text content, tool calls, and metadata

Tool message - Represents the outputs of **tool calls**

System message

A `SystemMessage` represent an initial set of instructions that primes the model's behavior. You can use a system message to set the tone, define the model's role, and establish guidelines for responses.

Basic instructions

```
import { SystemMessage, HumanMessage, AIMessage } from "langchain";

const systemMsg = new SystemMessage("You are a helpful coding assistant.");

const messages = [
  systemMsg,
  new HumanMessage("How do I create a REST API?"),
];
const response = await model.invoke(messages);
```

```
You are a senior TypeScript developer with expertise in web frameworks.
Always provide code examples and explain your reasoning.
Be concise but thorough in your explanations.
`);

const messages = [
  systemMsg,
  new HumanMessage("How do I create a REST API?"),
];
const response = await model.invoke(messages);
```

Human message

A `HumanMessage` represents user input and interactions. They can contain text, images, audio, files, and any other amount of multimodal **content**.

Text content

Message object

```
const response = await model.invoke([
  new HumanMessage("What is machine learning?"),
]);
```

String shortcut

```
const response = await model.invoke("What is machine learning?");
```

```
  content: "Hello!",
  name: "alice",
  id: "msg_123",
});
```

The `name` field behavior varies by provider – some use it for user identification, others ignore it. To check, refer to the model provider's **reference**.

AI message

An `AIMessage` represents the output of a model invocation. They can include multimodal data, tool calls, and provider-specific metadata that you can later access.

```
const response = await model.invoke("Explain AI");
console.log(typeof response);  // AIMessage
```

`AIMessage` objects are returned by the model when calling it, which contains all of the associated metadata in the response.

Providers weigh/contextualize types of messages differently, which means it is sometimes helpful to manually create a new `AIMessage` object and insert it into the message history as if it came from the model.

```
const messages = [
  new SystemMessage("You are a helpful assistant"),
  new HumanMessage("Can you help me?"),
  aiMsg,  // Insert as if it came from the model
  new HumanMessage("Great! What's 2+2?")
]

const response = await model.invoke(messages);
```

Attributes

`text` `string`

The text content of the message.

`content` `string | ContentBlock[]`

The raw content of the message.

`content_blocks` `ContentBlock.Standard[]`

The standardized content blocks of the message. (See content)

`tool_calls` `ToolCall[] | None`

The tool calls made by the model.

Empty if no tools are called.

`id` `string`

A unique identifier for the message (either automatically generated by LangChain or returned in

The usage metadata of the message, which can contain token counts when available. See `UsageMetadata`.

`response_metadata` `ResponseMetadata | None`

The response metadata of the message.

Tool calls

When models make **tool calls**, they're included in the `AIMessage`:

```
const modelWithTools = model.bindTools([getWeather]);
const response = await modelWithTools.invoke("What's the weather in Paris?");

for (const toolCall of response.tool_calls) {
  console.log(`Tool: ${toolCall.name}`);
  console.log(`Args: ${toolCall.args}`);
  console.log(`ID: ${toolCall.id}`);
}
```

Other structured data, such as reasoning or citations, can also appear in message **content**.

Token usage

An `AIMessage` can hold token counts and other usage metadata in its `usage_metadata` field:

```
const response = await model.invoke("Hello!");
console.log(response.usage_metadata);
```

```
{
  "output_tokens": 304,
  "input_tokens": 8,
  "total_tokens": 312,
  "input_token_details": {
    "cache_read": 0
  },
  "output_token_details": {
    "reasoning": 256
  }
}
```

See `UsageMetadata` for details.

Streaming and chunks

During streaming, you'll receive `AIMessageChunk` objects that can be combined into a full message object:

```
import { AIMessageChunk } from "langchain";

let finalChunk: AIMessageChunk | undefined;
for (const chunk of chunks) {
  finalChunk = finalChunk ? finalChunk.concat(chunk) : chunk;
}
```

Tool message

For models that support **tool calling**, AI messages can contain tool calls. Tool messages are used to pass the results of a single tool execution back to the model.

Tools can generate `ToolMessage` objects directly. Below, we show a simple example. Read more in the **tools guide**.

```
  tool_calls: [{
    name: "get_weather",
    args: { location: "San Francisco" },
    id: "call_123"
  }]
});

const toolMessage = new ToolMessage({
  content: "Sunny, 72°F",
  tool_call_id: "call_123"
});

const messages = [
  new HumanMessage("What's the weather in San Francisco?"),
  aiMessage,  // Model's tool call
  toolMessage,  // Tool execution result
];

const response = await model.invoke(messages);  // Model processes the result
```

Attributes

`content` `string` required

The stringified output of the tool call.

`tool_call_id` `string` required

The ID of the tool call that this message is responding to. Must match the ID of the tool call in the `AIMessage`.

`.ame` `string` required

Additional data not sent to the model but can be accessed programmatically.

The `artifact` field stores supplementary data that won't be sent to the model but can be accessed programmatically. This is useful for storing raw results, debugging information, or data for downstream processing without cluttering the model's context.

Example: Using artifact for retrieval metadata

For example, a **retrieval** tool could retrieve a passage from a document for reference by a model. Where message `content` contains text that the model will reference, an `artifact` can contain document identifiers or other metadata that an application can use (e.g., to render a page). See example below:

```
import { ToolMessage } from "langchain";

// Artifact available downstream
const artifact = { document_id: "doc_123", page: 0 };

const toolMessage = new ToolMessage({
  content: "It was the best of times, it was the worst of times."
  tool_call_id: "call_123",
  name: "search_books",
  artifact
});
```

See the **RAG tutorial** for an end-to-end example of building retrieval **agents** with LangChain.

You can think of a message's content as the payload of data that gets sent to the model. Messages have a `content` attribute that is loosely-typed, supporting strings and lists of untyped objects (e.g., dictionaries). This allows support for provider-native structures directly in LangChain chat models, such as **multimodal** content and other data.

Separately, LangChain provides dedicated content types for text, reasoning, citations, multi-modal data, server-side tool calls, and other message content. See **content blocks** below.

LangChain chat models accept message content in the `content` attribute.

This may contain either:

1. A string
2. A list of content blocks in a provider-native format
3. A list of **LangChain's standard content blocks**

See below for an example using **multimodal** inputs:

```
// Provider-native format (e.g., OpenAI)
const humanMessage = new HumanMessage({
  content: [
    { type: "text", text: "Hello, how are you?" },
    {
      type: "image_url",
      image_url: { url: "https://example.com/image.jpg" },
    },
  ],
});

// List of standard content blocks
const humanMessage = new HumanMessage({
  contentBlocks: [
    { type: "text", text: "Hello, how are you?" },
    { type: "image", url: "https://example.com/image.jpg" },
  ],
});
```

Standard content blocks

LangChain provides a standard representation for message content that works across providers.

Message objects implement a `contentBlocks` property that will lazily parse the `content` attribute into a standard, type-safe representation. For example, messages generated from `ChatAnthropic` or `ChatOpenAI` will include `thinking` or `reasoning` blocks in the format of the respective provider, but can be lazily parsed into a consistent `ReasoningContentBlock` representation:

opic OpenAI

```
  content: [
    {
      "type": "thinking",
      "thinking": "...",
      "signature": "WaUjzkyp...",
    },
    {
      "type":"text",
      "text": "...",
      "id": "msg_abc123",
    },
  ],
  response_metadata: { model_provider: "anthropic" },
});

console.log(message.contentBlocks);
```

See the **integrations guides** to get started with the inference provider of your choice.

To do this, you can set the `LC_OUTPUT_VERSION` environment variable to `v1`. Or, initialize any chat model with `outputVersion: "v1"`:

```
import { initChatModel } from "langchain";

const model = await initChatModel(
  "gpt-5-nano",
  { outputVersion: "v1" }
);
```

Multimodal

Multimodality refers to the ability to work with data that comes in different forms, such as text, audio, images, and video. LangChain includes standard types for these data that can be used across providers.

Chat models can accept multimodal data as input and generate it as output. Below we show short examples of input messages featuring multimodal data.

Extra keys can be included top-level in the content block or nested in `"extras": {"key": value}`.

OpenAI and AWS Bedrock Converse, for example, require a filename for PDFs. See the provider page for your chosen model for specifics.

```
    { type: "text", text: "Describe the content of this image." },
    {
      type: "image",
      source_type: "url",
      url: "https://example.com/path/to/image.jpg"
    },
  ],
});

// From base64 data
const message = new HumanMessage({
  content: [
    { type: "text", text: "Describe the content of this image." },
    {
      type: "image",
      source_type: "base64",
      data: "AAAAIGZ0eXBtcDQyAAAAAGlzb21tcDQyAAACAGlzb2...",
    },
  ],
});

// From provider-managed File ID
const message = new HumanMessage({
  content: [
    { type: "text", text: "Describe the content of this image." },
    { type: "image", source_type: "id", id: "file-abc123" },
  ],
});
```

⚠ Not all models support all file types. Check the model provider's **reference** for supported formats and size limits.

of the following block types:

Core

ContentBlock.Text

Purpose: Standard text output

`type` `string` required

Always `"text"`

`text` `string` required

The text content

`annotations` `Citation[]`

List of annotations for the text

Example:

```
{
    type: "text",
    text: "Hello world",
    annotations: []
}
```

ContentBlock.Reasoning

Always `"reasoning"`

`reasoning` `string` `required`

The reasoning content

Example:

```
{
    type: "reasoning",
    reasoning: "The user is asking about..."
}
```

Multimodal

ContentBlock.Multimodal.Image

Purpose: Image data

`type` `string` `required`

Always `"image"`

`url` `string`

URL pointing to the image location.

`data` `string`

Reference to the image in an external file storage system (e.g., OpenAI or Anthropic's Files API).

`mimeType` `string`

Image **MIME type** (e.g., `image/jpeg` , `image/png`). Required for base64 data.

ContentBlock.Multimodal.Audio

Purpose: Audio data

`type` `string` `required`

Always `"audio"`

`url` `string`

URL pointing to the audio location.

`data` `string`

Base64-encoded audio data.

`fileId` `string`

Reference to the audio file in an external file storage system (e.g., OpenAI or Anthropic's Files API).

`mimeType` `string`

Audio **MIME type** (e.g., `audio/mpeg` , `audio/wav`). Required for base64 data.

`type` `string` `required`

Always `"video"`

`url` `string`

URL pointing to the video location.

`data` `string`

Base64-encoded video data.

`fileId` `string`

Reference to the video file in an external file storage system (e.g., OpenAI or Anthropic's Files API).

`mimeType` `string`

Video **MIME type** (e.g., `video/mp4` , `video/webm`). Required for base64 data.

ContentBlock.Multimodal.File

Purpose: Generic files (PDF, etc)

`type` `string` `required`

Always `"file"`

LangChain Docs

`data` `string`

Base64-encoded file data.

`fileId` `string`

Reference to the file in an external file storage system (e.g., OpenAI or Anthropic's Files API).

`mimeType` `string`

File **MIME type** (e.g., `application/pdf`). Required for base64 data.

ContentBlock.Multimodal.PlainText

Purpose: Document text (`.txt` , `.md`)

`type` `string` `required`

Always `"text-plain"`

`text` `string` `required`

The text content

`title` `string`

Title of the text content

`mimeType` `string`

Tool Calling

ContentBlock.Tools.ToolCall

Purpose: Function calls

`type` `string` `required`

Always `"tool_call"`

`name` `string` `required`

Name of the tool to call

`args` `object` `required`

Arguments to pass to the tool

`id` `string` `required`

Unique identifier for this tool call

Example:

```
{
    type: "tool_call",
    name: "search",
    args: { query: "weather" },
    id: "call_123"
}
```

LangChain Docs

Core components › **Messages**

`type` `string` `required`

Always `"tool_call_chunk"`

`name` `string`

Name of the tool being called

`args` `string`

Partial tool arguments (may be incomplete JSON)

`id` `string`

Tool call identifier

`index` `number | string` `required`

Position of this chunk in the stream

ContentBlock.Tools.InvalidToolCall

Purpose: Malformed calls

`type` `string` `required`

Always `"invalid_tool_call"`

`name` `string`

Name of the tool that failed to be called `error` `string` `required`

Description of what went wrong

Common errors: Invalid JSON, missing required fields

Server-Side Tool Execution

ContentBlock.Tools.ServerToolCall

Purpose: Tool call that is executed server-side.

`type` `string` `required`

Always `"server_tool_call"`

`id` `string` `required`

An identifier associated with the tool call.

`name` `string` `required`

The name of the tool to be called.

`args` `string` `required`

Partial tool arguments (may be incomplete JSON)

ContentBlock.Tools.ServerToolCallChunk

Always `"server_tool_call_chunk"`

`id` `string`

An identifier associated with the tool call.

`name` `string`

Name of the tool being called

`args` `string`

Partial tool arguments (may be incomplete JSON)

`index` `number | string`

Position of this chunk in the stream

ContentBlock.Tools.ServerToolResult

Purpose: Search results

`type` `string` `required`

Always `"server_tool_result"`

`tool_call_id` `string` `required`

Identifier of the corresponding server tool call.

`id` `string`

Execution status of the server-side tool. `"success"` or `"error"` .

`output`

Output of the executed tool.

Provider-Specific Blocks

ContentBlock.NonStandard

Purpose: Provider-specific escape hatch

`type` `string` `required`

Always `"non_standard"`

`value` `object` `required`

Provider-specific data structure

Usage: For experimental or provider-unique features

Additional provider-specific content types may be found within the **reference documentation** of each model provider.

Each of these content blocks mentioned above are indvidually addressable as types when importing the `ContentBlock` type.

```
    type: "text",
    text: "Hello world",
}

// Image block
const imageBlock: ContentBlock.Multimodal.Image = {
    type: "image",
    url: "https://example.com/image.png",
    mimeType: "image/png",
}
```

View the canonical type definitions in the **API reference**.

Content blocks were introduced as a new property on messages in LangChain v1 to standardize content formats across providers while maintaining backward compatibility with existing code.

Content blocks are not a replacement for the `content` property, but rather a new property that can be used to access the content of a message in a standardized format.

Use with chat models

Chat models accept a sequence of message objects as input and return an `AIMessage` as output. Interactions are often stateless, so that a simple conversational loop involves invoking a model with a growing list of messages.

Refer to the below guides to learn more:

- Built-in features for **persisting and managing conversation histories**
- Strategies for managing context windows, including **trimming and summarizing messages**

LangChain Docs

Core components › **Messages**

Connect these docs to Claude, VSCode, and more via MCP for real-time answers.

Was this page helpful? Yes No

LangChain Docs

Resources

Forum

Changelog

LangChain Academy

Trust Center

Company

About

Careers

Blog

Powered by mintlify

LangChain Docs

Core components › **Messages**