Core components

Models

Copy page

LLMs are powerful AI tools that can interpret and generate text like humans. They're versatile enough to write content, translate languages, summarize, and answer questions without needing specialized training for each task.

In addition to text generation, many models support:

- **Tool calling** - calling external tools (like databases queries or API calls) and use results in their responses.
- **Structured output** - where the model's response is constrained to follow a defined format.
- **Multimodality** - process and return data other than text, such as images, audio, and video.
- **Reasoning** - models perform multi-step reasoning to arrive at a conclusion.

Models are the reasoning engine of **agents**. They drive the agent's decision-making process, determining which tools to call, how to interpret results, and when to provide a final answer.

The quality and capabilities of the model you choose directly impact your agent's baseline reliability and performance. Different models excel at different tasks - some are better at following complex instructions, others at structured reasoning, and some support larger context windows for handling more information.

LangChain's standard model interfaces give you access to many different provider integrations, which makes it easy to experiment with and switch between models to find the best fit for your use case.

Basic usage

Models can be utilized in two ways:

1. **With agents** - Models can be dynamically specified when creating an **agent**.
2. **Standalone** - Models can be called directly (outside of the agent loop) for tasks like text generation, classification, or extraction without the need for an agent framework.

The same model interface works in both contexts, which gives you the flexibility to start simple and scale up to more complex agent-based workflows as needed.

Initialize a model

The easiest way to get started with a standalone model in LangChain is to use `initChatModel` to initialize one from a **chat model provider** of your choice (examples below):

OpenAI **Anthropic** **Azure** **Google Gemini** **Bedrock Converse**

👉 Read the **OpenAI chat model integration docs**

npm pnpm yarn bun

```
npm install @langchain/openai
```

LangChain Docs

LangChain Docs

LangChain Docs

initChatModel Model Class

parameters.

```
const model = await initChatModel("gpt-4.1");
```

Supported models

LangChain supports all major model providers, including OpenAI, Anthropic, Google, Azure, AWS Bedrock, and more. Each provider offers a variety of models with different capabilities. For a full list of supported models in LangChain, see the **integrations page**.

Key methods

Invoke
The model takes messages as input and outputs messages after generating a complete response.

Stream
Invoke the model, but stream the output as it is generated in real-time.

Batch
Send multiple requests to a model in a batch for more efficient processing.

ⓘ In addition to chat models, LangChain provides support for other adjacent technologies, such as embedding models and vector stores. See the **integrations page** for details.

Parameters

at model takes parameters that can be used to configure its behavior. The full set of ported parameters varies by model and provider, but standard ones include:

`apiKey` `string`

The key required for authenticating with the model's provider. This is usually issued when you sign up for access to the model. Often accessed by setting an environment variable.

`temperature` `number`

Controls the randomness of the model's output. A higher number makes responses more creative; lower ones make them more deterministic.

`maxTokens` `number`

Limits the total number of tokens in the response, effectively controlling how long the output can be.

`timeout` `number`

The maximum time (in seconds) to wait for a response from the model before canceling the request.

`maxRetries` `number`

The maximum number of attempts the system will make to resend a request if it fails due to issues like network timeouts or rate limits.

Using `initChatModel` , pass these parameters as inline parameters:

Initialize using model parameters

```
const model = await initChatModel(
    "claude-sonnet-4-5-20250929",
    { temperature: 0.7, timeout: 30, max_tokens: 1000 }
```

To find all the parameters supported by a given chat model, head to the **chat model integrations** page.

Invocation

A chat model must be invoked to generate an output. There are three primary invocation methods, each suited to different use cases.

Invoke

The most straightforward way to call a model is to use `invoke()` with a single message or a list of messages.

Single message

```
const response = await model.invoke("Why do parrots have colorful feathers?");
console.log(response);
```

A list of messages can be provided to a chat model to represent conversation history. Each message has a role that models use to indicate who sent the message in the conversation.

See the **messages** guide for more detail on roles, types, and content.

```
  { role: "user", content: "Translate: I love programming." },
  { role: "assistant", content: "J'adore la programmation." },
  { role: "user", content: "Translate: I love building applications." },
];

const response = await model.invoke(conversation);
console.log(response);  // AIMessage("J'adore créer des applications.")
```

Message objects

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

const conversation = [
  new SystemMessage("You are a helpful assistant that translates English to Fre
  new HumanMessage("Translate: I love programming."),
  new AIMessage("J'adore la programmation."),
  new HumanMessage("Translate: I love building applications."),
];

const response = await model.invoke(conversation);
console.log(response);  // AIMessage("J'adore créer des applications.")
```

ⓘ If the return type of your invocation is a string, ensure that you are using a chat model as opposed to a LLM. Legacy, text-completion LLMs return strings directly. LangChain chat models are prefixed with “Chat”, e.g., `ChatOpenAI`(/oss/integrations/chat/openai).

Stream

Most models can stream their output content while it is being generated. By displaying output progressively, streaming significantly improves user experience, particularly for longer responses.

```
const stream = await model.stream("Why do parrots have colorful feathers?");
for await (const chunk of stream) {
  console.log(chunk.text)
}
```

As opposed to `invoke()`, which returns a single `AIMessage` after the model has finished generating its full response, `stream()` returns multiple `AIMessageChunk` objects, each containing a portion of the output text. Importantly, each chunk in a stream is designed to be gathered into a full message via summation:

Construct AIMessage

```
let full: AIMessageChunk | null = null;
for await (const chunk of stream) {
  full = full ? full.concat(chunk) : chunk;
  console.log(full.text);
}

// The
// The sky
// The sky is
// The sky is typically
// The sky is typically blue
// ...

console.log(full.contentBlocks);
// [{"type": "text", "text": "The sky is typically blue..."}]
```

The resulting message can be treated the same as a message that was generated with `invoke()` – for example, it can be aggregated into a message history and passed back to model as conversational context.

Advanced streaming topics

Streaming events

LangChain chat models can also stream semantic events using [`streamEvents()`][BaseChatModel.streamEvents].

This simplifies filtering based on event types and other metadata, and will aggregate the full message in the background. See below for an example.

```
const stream = await model.streamEvents("Hello");
for await (const event of stream) {
    if (event.event === "on_chat_model_start") {
        console.log(`Input: ${event.data.input}`);
    }
    if (event.event === "on_chat_model_stream") {
        console.log(`Token: ${event.data.chunk.text}`);
    }
    if (event.event === "on_chat_model_end") {
        console.log(`Full message: ${event.data.output.text}`);
    }
}
```

```
Token: !
Token:  How
Token:  can
Token:  I
...
Full message: Hi there! How can I help today?
```

See the `streamEvents()` reference for event types and other details.

"Auto-streaming" chat models

LangChain simplifies streaming from chat models by automatically enabling streaming mode in certain cases, even when you're not explicitly calling the streaming methods. This is particularly useful when you use the non-streaming invoke method but still want to stream the entire application, including intermediate results from the chat model.

In **LangGraph agents**, for example, you can call `model.invoke()` within nodes, but LangChain will automatically delegate to streaming if running in a streaming mode.

How it works

When you `invoke()` a chat model, LangChain will automatically switch to an internal streaming mode if it detects that you are trying to stream the overall application. The result of the invocation will be the same as far as the code that was using invoke is concerned; however, while the chat model is being streamed, LangChain will take care of invoking `on_llm_new_token` events in LangChain's callback system.

Callback events allow LangGraph `stream()` and `streamEvents()` to

Batch

Batching a collection of independent requests to a model can significantly improve performance and reduce costs, as the processing can be done in parallel:

Batch

```
const responses = await model.batch([
  "Why do parrots have colorful feathers?",
  "How do airplanes fly?",
  "What is quantum computing?",
  "Why do parrots have colorful feathers?",
  "How do airplanes fly?",
  "What is quantum computing?",
]);
for (const response of responses) {
  console.log(response);
}
```

Batch with max concurrency

```
model.batch(
  listOfInputs,
  {
    maxConcurrency: 5,  // Limit to 5 parallel calls
  }
)
```

See the `RunnableConfig` reference for a full list of supported attributes.

For more details on batching, see the **reference**.

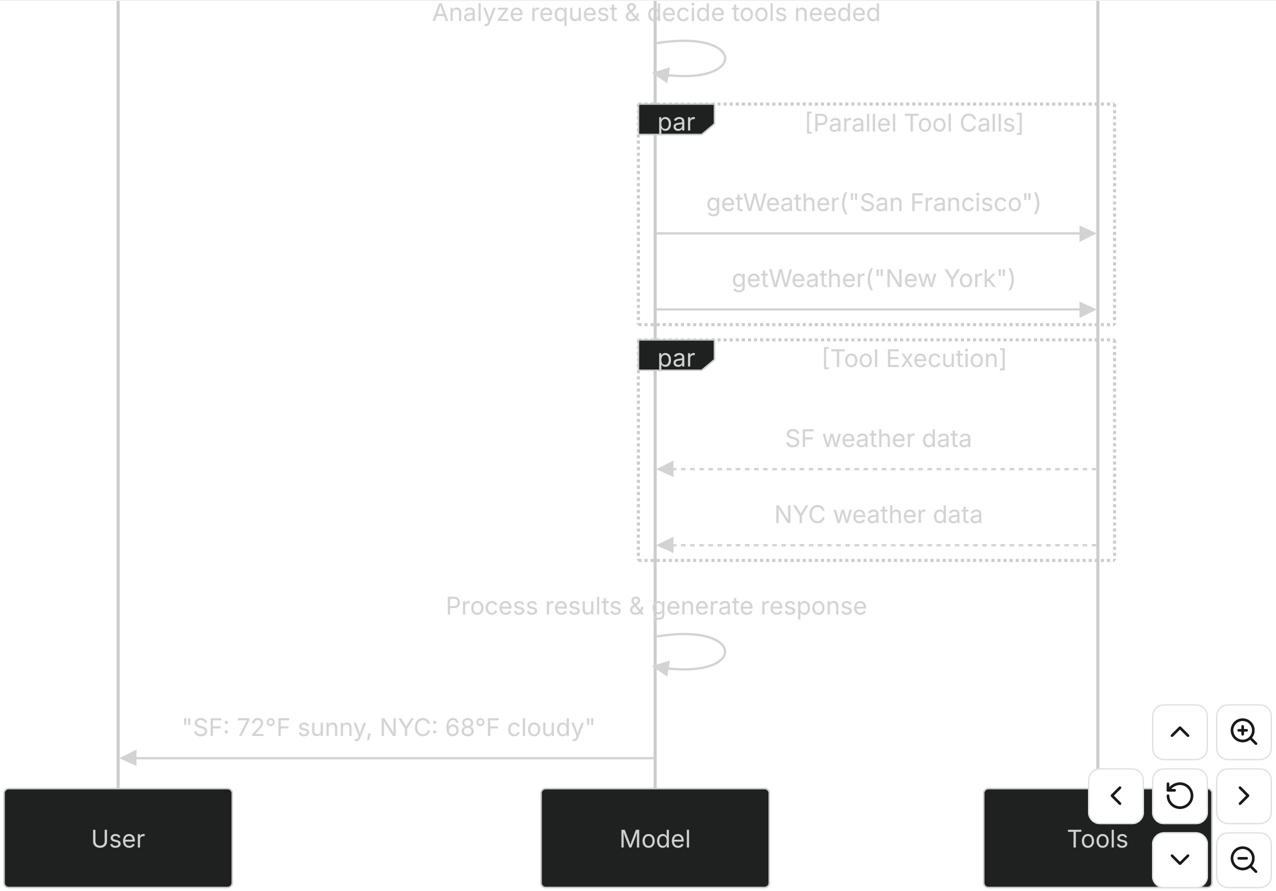
Tool calling

Models can request to call tools that perform tasks such as fetching data from a database, searching the web, or running code. Tools are pairings of:

1. A schema, including the name of the tool, a description, and/or argument definitions (often a JSON schema)
2. A function or coroutine to execute.

ⓘ You may hear the term “function calling”. We use this interchangeably with “tool calling”.

Here’s the basic tool calling flow between a user and a model:


Analyze request & decide tools needed
par
[Parallel Tool Calls]
getWeather("San Francisco")
getWeather("New York")
par
[Tool Execution]
SF weather data
NYC weather data
Process results & generate response
"SF: 72°F sunny, NYC: 68°F cloudy"
User
Model
Tools

To make tools that you have defined available for use by a model, you must bind them using `bindTools`. In subsequent invocations, the model can choose to call any of the bound tools as needed.

Some model providers offer built-in tools that can be enabled via model or invocation parameters (e.g. `ChatOpenAI`, `ChatAnthropic`). Check the respective **provider reference** for details.

See the **tools guide** for details and other options for creating tools.

```
import { ChatOpenAI } from "@langchain/openai";

const getWeather = tool(
  (input) => `It's sunny in ${input.location}.`,
  {
    name: "get_weather",
    description: "Get the weather at a location.",
    schema: z.object({
      location: z.string().describe("The location to get the weather for"),
    }),
  },
);

const model = new ChatOpenAI({ model: "gpt-4o" });
const modelWithTools = model.bindTools([getWeather]);

const response = await modelWithTools.invoke("What's the weather like in Boston
const toolCalls = response.tool_calls || [];
for (const tool_call of toolCalls) {
  // View tool calls made by the model
  console.log(`Tool: ${tool_call.name}`);
  console.log(`Args: ${tool_call.args}`);
}
```

When binding user-defined tools, the model's response includes a **request** to execute a tool. When using a model separately from an agent, it is up to you to execute the requested tool and return the result back to the model for use in subsequent reasoning. When using an agent, the agent loop will handle the tool execution loop for you.

Below, we show some common ways you can use tool calling.

Tool execution loop

Here’s a simple example of how to do this:

Tool execution loop

```
// Bind (potentially multiple) tools to the model
const modelWithTools = model.bindTools([get_weather])

// Step 1: Model generates tool calls
const messages = [{"role": "user", "content": "What's the weather in Bost
const ai_msg = await modelWithTools.invoke(messages)
messages.push(ai_msg)

// Step 2: Execute tools and collect results
for (const tool_call of ai_msg.tool_calls) {
    // Execute the tool with the generated arguments
    const tool_result = await get_weather.invoke(tool_call)
    messages.push(tool_result)
}

// Step 3: Pass results back to model for final response
const final_response = await modelWithTools.invoke(messages)
console.log(final_response.text)
// "The current weather in Boston is 72°F and sunny."
```

Each `ToolMessage` returned by the tool includes a `tool_call_id` that matches the original tool call, helping the model correlate results with requests.

Forcing tool calls

By default, the model has the freedom to choose which bound tool to use based on e user’s input. However, you might want to force choosing a tool, ensuring the model uses either a particular tool or **any** tool from a given list:

Parallel tool calls

Many models support calling multiple tools in parallel when appropriate. This allows the model to gather information from different sources simultaneously.

Parallel tool calls

```
const modelWithTools = model.bind_tools([get_weather])

const response = await modelWithTools.invoke(
    "What's the weather in Boston and Tokyo?"
)


// The model may generate multiple tool calls
console.log(response.tool_calls)
// [
//   { name: 'get_weather', args: { location: 'Boston' }, id: 'call_1' },
//   { name: 'get_time', args: { location: 'Tokyo' }, id: 'call_2' }
// ]


// Execute all tools (can be done in parallel with async)
const results = []
for (const tool_call of response.tool_calls || []) {
    if (tool_call.name === 'get_weather') {
        const result = await get_weather.invoke(tool_call)
        results.push(result)
    }
}
```

Most models supporting tool calling enable parallel tool calls by default. Some (including **OpenAI** and **Anthropic**) allow you to disable this feature. To do this, set `parallel_tool_calls=False` :

```
model.bind_tools([get_weather], parallel_tool_calls=F
```

Streaming tool calls

When streaming responses, tool calls are progressively built through `ToolCallChunk`. This allows you to see tool calls as they're being generated rather than waiting for the complete response.

```
)
for await (const chunk of stream) {
    // Tool call chunks arrive progressively
    if (chunk.tool_call_chunks) {
        for (const tool_chunk of chunk.tool_call_chunks) {
        console.log(`Tool: ${tool_chunk.get('name', '')}`)
        console.log(`Args: ${tool_chunk.get('args', '')}`)
        }
    }
}

// Output:
// Tool: get_weather
// Args:
// Tool:
// Args: {"loc
// Tool:
// Args: ation": "BOS"}
// Tool: get_time
// Args:
// Tool:
// Args: {"timezone": "Tokyo"}
```

You can accumulate chunks to build complete tool calls:

Accumulate tool calls

```
let full: AIMessageChunk | null = null
const stream = await modelWithTools.stream("What's the weather in Boston
for await (const chunk of stream) {
    full = full ? full.concat(chunk) : chunk
    console.log(full.contentBlocks)
}
```

Models can be requested to provide their response in a format matching a given schema. This is useful for ensuring the output can be easily parsed and used in subsequent processing. LangChain supports multiple schema types and methods for enforcing structured output.

To learn about structured output, see **Structured output**.

Zod JSON Schema

A **zod schema** is the preferred method of defining an output schema. Note that when a zod schema is provided, the model output will also be validated against the schema using zod's parse methods.

```
import * as z from "zod";

const Movie = z.object({
  title: z.string().describe("The title of the movie"),
  year: z.number().describe("The year the movie was released"),
  director: z.string().describe("The director of the movie"),
  rating: z.number().describe("The movie's rating out of 10"),
});

const modelWithStructure = model.withStructuredOutput(Movie);

const response = await modelWithStructure.invoke("Provide details about the mov
console.log(response);
// {
//   title: "Inception",
//   year: 2010,
//   director: "Christopher Nolan",
//   rating: 8.8,
// }
```

Include raw: Use `includeRaw: true` to get both the parsed output and the raw `AIMessage`

Validation: Zod models provide automatic validation, while JSON Schema requires manual validation

See your **provider's integration page** for supported methods and configuration options.

include_raw=True when calling with_structured_output:

```
import * as z from "zod";

const Movie = z.object({
  title: z.string().describe("The title of the movie"),
  year: z.number().describe("The year the movie was released"),
  director: z.string().describe("The director of the movie"),
  rating: z.number().describe("The movie's rating out of 10"),
  title: z.string().describe("The title of the movie"),
  year: z.number().describe("The year the movie was released"),
  director: z.string().describe("The director of the movie"),
  rating: z.number().describe("The movie's rating out of 10"),
});

const modelWithStructure = model.withStructuredOutput(Movie, { includeRaw

const response = await modelWithStructure.invoke("Provide details about t
console.log(response);
// {
//   raw: AIMessage { ... },
//   parsed: { title: "Inception", ... }
// }
```

Example: Nested structures

Schemas can be nested:

```
  name: str
  role: z.string(),
});

const MovieDetails = z.object({
  title: z.string(),
  year: z.number(),
  cast: z.array(Actor),
  genres: z.array(z.string()),
  budget: z.number().nullable().describe("Budget in millions USD"),
});

const modelWithStructure = model.withStructuredOutput(MovieDetails);
```

Advanced topics

Model profiles

ⓘ Model profiles require `langchain>=1.1` .

LangChain chat models can expose a dictionary of supported features and capabilities through a `.profile` property:

```
//   reasoningOutput: true,
//   toolCalling: true,
//   ...
// }
```

Refer to the full set of fields in the **API reference**.

Much of the model profile data is powered by the **models.dev** project, an open source initiative that provides model capability data. This data is augmented with additional fields for purposes of use with LangChain. These augmentations are kept aligned with the upstream project as it evolves.

Model profile data allow applications to work around model capabilities dynamically. For example:

1. **Summarization middleware** can trigger summarization based on a model's context window size.
2. **Structured output** strategies in `createAgent` can be inferred automatically (e.g., by checking support for native structured output features).
3. Model inputs can be gated based on supported **modalities** and maximum input tokens.

Modify profile data

Model profile data can be changed if it is missing, stale, or incorrect.

Option 1 (quick fix)

You can instantiate a chat model with any valid profile:

```
structuredOutput: true,
// ...
};
const model = initChatModel("...", { profile: customProfile });
```

Option 2 (fix data upstream)

The primary source for the data is the **models.dev** project. These data are merged with additional fields and overrides in LangChain **integration packages** and are shipped with those packages.

Model profile data can be updated through the following process:

1. (If needed) update the source data at **models.dev** through a pull request to its **repository on GitHub**.
2. (If needed) update additional fields and overrides in `langchain-<package>/profiles.toml` through a pull request to the LangChain **integration package**.

⚠ Model profiles are a beta feature. The format of a profile is subject to change.

Multimodal

Certain models can process and return non-textual data such as images, audio, and video. You can pass non-textual data to a model by providing **content blocks**.

3. Any format that is native to that specific provider (e.g., Anthropic models accept Anthropic native format)

See the **multimodal section** of the messages guide for details.

Some models can return multimodal data as part of their response. If invoked to do so, the resulting `AIMessage` will have content blocks with multimodal types.

Multimodal output

```
const response = await model.invoke("Create a picture of a cat");
console.log(response.contentBlocks);
// [
//   { type: "text", text: "Here's a picture of a cat" },
//   { type: "image", data: "...", mimeType: "image/jpeg" },
// ]
```

See the **integrations page** for details on specific providers.

Reasoning

Many models are capable of performing multi-step reasoning to arrive at a conclusion. This involves breaking down complex problems into smaller, more manageable steps.

If supported by the underlying model, you can surface this reasoning process to better understand how the model arrived at its final answer.

```
    const reasoningSteps = chunk.contentBlocks.filter(b => b.type === "reasoni
    console.log(reasoningSteps.length > 0 ? reasoningSteps : chunk.text);
}
```

Depending on the model, you can sometimes specify the level of effort it should put into reasoning. Similarly, you can request that the model turn off reasoning entirely. This may take the form of categorical "tiers" of reasoning (e.g., `'low'` or `'high'`) or integer token budgets.

For details, see the **integrations page** or **reference** for your respective chat model.

Local models

LangChain supports running models locally on your own hardware. This is useful for scenarios where either data privacy is critical, you want to invoke a custom model, or when you want to avoid the costs incurred when using a cloud-based model.

Ollama is one of the easiest ways to run chat and embedding models locally.

Prompt caching

Many providers offer prompt caching features to reduce latency and cost on repeat processing of the same tokens. These features can be **implicit** or **explicit**:

- **Implicit prompt caching:** providers will automatically pass on cost savings if a request hits a cache. Examples: **OpenAI** and **Gemini**.
- **Explicit caching:** providers allow you to manually indicate cache points for greater control or to guarantee cost savings. Examples:
    - `ChatOpenAI` (via `prompt_cache_key`)
    - Anthropic's `AnthropicPromptCachingMiddleware`

Prompt caching is often only engaged above a minimum input token threshold. See **provider pages** for details.

Cache usage will be reflected in the **usage metadata** of the model response.

Server-side tool use

Some providers support server-side **tool-calling** loops: models can interact with web search, code interpreters, and other tools and analyze the results in a single conversational turn.

If a model invokes a tool server-side, the content of the response message will include content representing the invocation and result of the tool. Accessing the **content blocks** of the response will return the server-side tool calls and results in a provider-agnostic format:

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

const model = await initChatModel("gpt-4.1-mini");
const modelWithTools = model.bindTools([{ type: "web_search" }])

const message = await modelWithTools.invoke("What was a positive news story fro
console.log(message.contentBlocks);
```

This represents a single conversational turn; there are no associated **ToolMessage** objects that need to be passed in as in client-side **tool-calling**.

See the **integration page** for your given provider for available tools and usage details.

e URL or proxy

Base URL

Many model providers offer OpenAI-compatible APIs (e.g., Together AI, vLLM). You can use `initChatModel` with these providers by specifying the appropriate `base_url` parameter:

```
model = initChatModel(
    "MODEL_NAME",
    {
        modelProvider: "openai",
        baseUrl: "BASE_URL",
        apiKey: "YOUR_API_KEY",
    }
)
```

ⓘ When using direct chat model class instantiation, the parameter name may vary by provider. Check the respective **reference** for details.

Log probabilities

Certain models can be configured to return token-level log probabilities representing the likelihood of a given token by setting the `logprobs` parameter when initializing the model:

```
const responseMessage = await model.invoke("Why do parrots talk?");

responseMessage.response_metadata.logprobs.content.slice(0, 5);
```

Token usage

A number of model providers return token usage information as part of the invocation response. When available, this information will be included on the `AIMessage` objects produced by the corresponding model. For more details, see the **messages** guide.

> ⓘ Some provider APIs, notably OpenAI and Azure OpenAI chat completions, require users opt-in to receiving token usage data in streaming contexts. See the **streaming usage metadata** section of the integration guide for details.

Invocation config

When invoking a model, you can pass additional configuration through the `config` parameter using a `RunnableConfig` object. This provides run-time control over execution behavior, callbacks, and metadata tracking.

Common configuration options include:

```
    {
        runName: "joke_generation",      // Custom name for this run
        tags: ["humor", "demo"],          // Tags for categorization
        metadata: {"user_id": "123"},     // Custom metadata
        callbacks: [my_callback_handler], // Callback handlers
    }
)
```

These configuration values are particularly useful when:

- Debugging with LangSmith tracing
- Implementing custom logging or monitoring
- Controlling resource usage in production
- Tracking invocations across complex pipelines

Key configuration attributes

`runName` `string`

Identifies this specific invocation in logs and traces. Not inherited by sub-calls.

`tags` `string[]`

Labels inherited by all sub-calls for filtering and organization in debugging tools.

`metadata` `object`

Custom key-value pairs for tracking additional context, inherited by all sub-calls.

`xConcurrency` `number`

Handlers for monitoring and responding to events during execution.

`recursion_limit` `number`

Maximum recursion depth for chains to prevent infinite loops in complex pipelines.

See full `RunnableConfig` reference for all supported attributes.

Edit this page on GitHub or **file an issue**.

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

Was this page helpful? Yes No

LangChain Docs

Core components › **Models**

LangChain Academy

Blog

Trust Center

Powered by mintlify