LangChain Docs

Core components › Agents

Core components

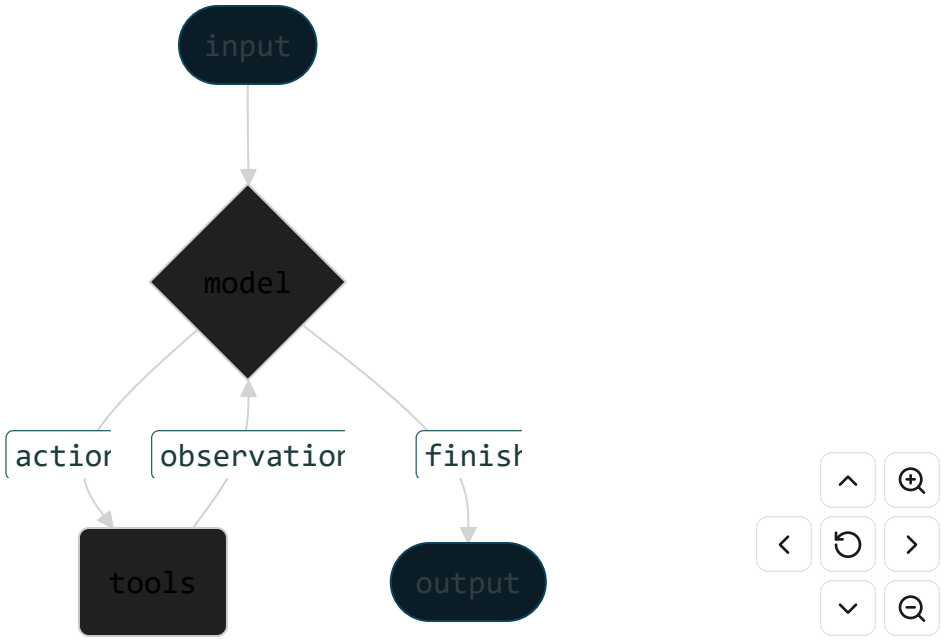
Agents

Copy page

Agents combine language models with **tools** to create systems that can reason about tasks, decide which tools to use, and iteratively work towards solutions.

`createAgent()` provides a production-ready agent implementation.

An LLM Agent runs tools in a loop to achieve a goal. An agent runs until a stop condition is met - i.e., when the model emits a final output or an iteration limit is reached.



`createAgent()` builds a **graph**-based agent runtime using **LangGraph**. A graph consists of nodes (steps) and edges (connections) that define how your agent processes information. The agent moves through this graph, executing nodes like the model node (which calls the model), the tools node (which executes tools), or middleware.

Learn more about the **Graph API**.

supporting both static and dynamic model selection.

Static model

Static models are configured once when creating the agent and remain unchanged throughout execution. This is the most common and straightforward approach.

To initialize a static model from a model identifier string:

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

const agent = createAgent({
  model: "openai:gpt-5",
  tools: []
});
```

Model identifier strings use the format `provider:model` (e.g. `"openai:gpt-5"`). You may want more control over the model configuration, in which case you can initialize a model instance directly using the provider package:

LangChain Docs

Core components › **Agents**

LangChain Docs

Core components › **Agents**

LangChain Docs

Core components › **Agents**

LangChain Docs

Core components › **Agents**

LangChain Docs

Core components › **Agents**

LangChain Docs

Core components › **Agents**

LangChain Docs

Core components › **Agents**

LangChain Docs

Core components › **Agents**

LangChain Docs

Core components › **Agents**

LangChain Docs

Core components › **Agents**

LangChain Docs

Core components › **Agents**

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

```
});

const agent = createAgent({
  model,
  tools: []
});
```

Model instances give you complete control over configuration. Use them when you need to set specific parameters like `temperature`, `max_tokens`, `timeouts`, or configure API keys, `base_url`, and other provider-specific settings. Refer to the **API reference** to see available params and methods on your model.

Dynamic model

Dynamic models are selected at runtime based on the current state and context. This enables sophisticated routing logic and cost optimization.

To use a dynamic model, create middleware with `wrapModelCall` that modifies the model in the request:

LangChain Docs

Core components › **Agents**

```
const advancedModel = new ChatOpenAI({ model: "gpt-4o" });

const dynamicModelSelection = createMiddleware({
  name: "DynamicModelSelection",
  wrapModelCall: (request, handler) => {
    // Choose model based on conversation complexity
    const messageCount = request.messages.length;

    return handler({
        ...request,
        model: messageCount > 10 ? advancedModel : basicModel,
    });
  },
});

const agent = createAgent({
  model: "gpt-4o-mini", // Base model (used when messageCount ≤ 10)
  tools,
  middleware: [dynamicModelSelection],
});
```

For more details on middleware and advanced patterns, see the **middleware documentation**.

> For model configuration details, see **Models**. For dynamic model selection patterns, see **Dynamic model in middleware**.

Tools

Tools give agents the ability to take actions. Agents go beyond simple model-only tool
ng by facilitating:

- Tool retry logic and error handling
- State persistence across tool calls

For more information, see **Tools**.

Defining tools

Pass a list of tools to the agent.

```
  ({ query }) => `Results for: ${query}`,
  {
    name: "search",
    description: "Search for information",
    schema: z.object({
      query: z.string().describe("The query to search for"),
    }),
  }
);

const getWeather = tool(
  ({ location }) => `Weather in ${location}: Sunny, 72°F`,
  {
    name: "get_weather",
    description: "Get weather information for a location",
    schema: z.object({
      location: z.string().describe("The location to get weather for"),
    }),
  }
);

const agent = createAgent({
  model: "gpt-4o",
  tools: [search, getWeather],
});
```

If an empty tool list is provided, the agent will consist of a single LLM node without tool-calling capabilities.

Tool error handling

To customize how tool errors are handled, use the `wrapToolCall` hook in a custom middleware:

```
  wrapToolCall: async (request, handler) => {
    try {
      return await handler(request);
    } catch (error) {
      // Return a custom error message to the model
      return new ToolMessage({
        content: `Tool error: Please check your input and try again.
(${error})`,
        tool_call_id: request.toolCall.id!,
      });
    }
  },
});

const agent = createAgent({
  model: "gpt-4o",
  tools: [
    /* ... */
  ],
  middleware: [handleToolErrors],
});
```

The agent will return a `ToolMessage` with the custom error message when a tool fails.

Tool use in the ReAct loop

Agents follow the ReAct (“Reasoning + Acting”) pattern, alternating between brief reasoning steps with targeted tool calls and feeding the resulting observations into subsequent decisions until they can deliver a final answer.

Example of ReAct loop

Prompt: Identify the current most popular wireless headphones and verify

```
Find the most popular wireless headphones right now and check if they're
```

Reasoning: "Popularity is time-sensitive, I need to use the provided search tool."

Acting: Call `search_products("wireless headphones")`

```
================================== Ai Message ==================
Tool Calls:
  search_products (call_abc123)
 Call ID: call_abc123
  Args:
    query: wireless headphones
```

```
================================= Tool Message =================

Found 5 products matching "wireless headphones". Top 5 results: WH-1000XM
```

Reasoning: "I need to confirm availability for the top-ranked item before answering."

Acting: Call `check_inventory("WH-1000XM5")`

```
================================== Ai Message ==================
Tool Calls:
  check_inventory (call_def456)
 Call ID: call_def456
  Args:
    product_id: WH-1000XM5
```

Reasoning: "I have the most popular model and its stock status. I can now answer the user's question."

Acting: Produce final answer

```
================================== Ai Message ==============:

I found wireless headphones (model WH-1000XM5) with 10 units in stock...
```

To learn more about tools, see **Tools**.

System prompt

You can shape how your agent approaches tasks by providing a prompt. The `systemPrompt` parameter can be provided as a string:

```
const agent = createAgent({
  model,
  tools,
  systemPrompt: "You are a helpful assistant. Be concise and accurate.",
});
```

When no `systemPrompt` is provided, the agent will infer its task from the messages directly.

`systemPrompt` parameter accepts either a `string` or a `SystemMessage`. Using a

```
import { SystemMessage, HumanMessage } from "@langchain/core/messages";

const literaryAgent = createAgent({
  model: "anthropic:claude-sonnet-4-5",
  systemPrompt: new SystemMessage({
    content: [
      {
        type: "text",
        text: "You are an AI assistant tasked with analyzing literary
works.",
      },
      {
        type: "text",
        text: "<the entire contents of 'Pride and Prejudice'>",
        cache_control: { type: "ephemeral" }
      }
    ]
  })
});

const result = await literaryAgent.invoke({
  messages: [new HumanMessage("Analyze the major themes in 'Pride and
Prejudice'.")]
});
```

The `cache_control` field with `{ type: "ephemeral" }` tells Anthropic to cache that content block, reducing latency and costs for repeated requests that use the same system prompt.

Dynamic system prompt

For more advanced use cases where you need to modify the system prompt based on runtime context or agent state, you can use **middleware**.

```
  userRole: z.enum(["expert", "beginner"]),
});

const agent = createAgent({
  model: "gpt-4o",
  tools: [/* ... */],
  contextSchema,
  middleware: [
    dynamicSystemPromptMiddleware<z.infer<typeof contextSchema>>((state,
runtime) => {
      const userRole = runtime.context.userRole || "user";
      const basePrompt = "You are a helpful assistant.";

      if (userRole === "expert") {
        return `${basePrompt} Provide detailed technical responses.`;
      } else if (userRole === "beginner") {
        return `${basePrompt} Explain concepts simply and avoid jargon.`;
      }
      return basePrompt;
    }),
  ],
});

// The system prompt will be set dynamically based on context
const result = await agent.invoke(
  { messages: [{ role: "user", content: "Explain machine learning" }] },
  { context: { userRole: "expert" } }
);
```

For more details on message types and formatting, see **Messages**. For comprehensive middleware documentation, see **Middleware**.

```
await agent.invoke({
  messages: [{ role: "user", content: "What's the weather in San Francisco?" }]
})
```

For streaming steps and / or tokens from the agent, refer to the **streaming** guide.

Otherwise, the agent follows the LangGraph **Graph API** and supports all associated methods, such as `stream` and `invoke`.

Advanced concepts

Structured output

In some situations, you may want the agent to return an output in a specific format. LangChain provides a simple, universal way to do this with the `responseFormat` parameter.

```
  name: z.string(),
  email: z.string(),
  phone: z.string(),
});

const agent = createAgent({
  model: "gpt-4o",
  responseFormat: ContactInfo,
});

const result = await agent.invoke({
  messages: [
    {
      role: "user",
      content: "Extract contact info from: John Doe, john@example.com, (555)
123-4567",
    },
  ],
});

console.log(result.structuredResponse);
// {
//   name: 'John Doe',
//   email: 'john@example.com',
//   phone: '(555) 123-4567'
// }
```

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

Memory

Agents maintain conversation history automatically through the message state. You can

LangChain Docs

Core components › **Agents**

```
import { z } from "zod/v4";
import { StateSchema, MessagesValue } from "@langchain/langgraph";
import { createAgent } from "langchain";

const CustomAgentState = new StateSchema({
  messages: MessagesValue,
  userPreferences: z.record(z.string(), z.string()),
});

const customAgent = createAgent({
  model: "gpt-4o",
  tools: [],
  stateSchema: CustomAgentState,
});
```

To learn more about memory, see **Memory**. For information on implementing long-term memory that persists across sessions, see **Long-term memory**.

Streaming

We've seen how the agent can be called with `invoke` to get a final response. If the agent executes multiple steps, this may take a while. To show intermediate progress, we can stream back messages as they occur.

```
      content: "Search for AI news and summarize the findings"
    }],
  },
  { streamMode: "values" }
);

for await (const chunk of stream) {
  // Each chunk contains the full state at that point
  const latestMessage = chunk.messages.at(-1);
  if (latestMessage?.content) {
    console.log(`Agent: ${latestMessage.content}`);
  } else if (latestMessage?.tool_calls) {
    const toolCallNames = latestMessage.tool_calls.map((tc) => tc.name);
    console.log(`Calling tools: ${toolCallNames.join(", ")}`);
  }
}
```

For more details on streaming, see **Streaming**.

Middleware

Middleware provides powerful extensibility for customizing agent behavior at different stages of execution. You can use middleware to:

- Process state before the model is called (e.g., message trimming, context injection)
- Modify or validate the model's response (e.g., guardrails, content filtering)
- Handle tool execution errors with custom logic
- Implement dynamic model selection based on state or context
- \dd custom logging, monitoring, or analytics

`afterModel` , and `wrapToolCall` , see **Middleware**.

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 › **Agents**

LangChain Docs

Core components › **Agents**