Core components › Streaming

Frontend

Copy page

Build generative UIs with real-time streaming from LangChain agents, LangGraph graphs, and custom APIs

The `useStream` React hook provides seamless integration with LangGraph streaming capabilities. It handles all the complexities of streaming, state management, and branching logic, letting you focus on building great generative UI experiences.

Key features:

- **Messages streaming** — Handle a stream of message chunks to form a complete message
- **Automatic state management** — for messages, interrupts, loading states, and errors
- **Conversation branching** — Create alternate conversation paths from any point in the chat history
- **UI-agnostic design** — Bring your own components and styling

Installation

Install the LangGraph SDK to use the `useStream` hook in your React application:

```
npm install @langchain/langgraph-sdk
```

Basic usage

```
function Chat() {
  const stream = useStream({
    assistantId: "agent",
    // Local development
    apiUrl: "http://localhost:2024",
    // Production deployment (LangSmith hosted)
    // apiUrl: "https://your-deployment.us.langgraph.app"
  });

  const handleSubmit = (message: string) => {
    stream.submit({
      messages: [
        { content: message, type: "human" }
      ],
    });
  };

  return (
    <div>
      {stream.messages.map((message, idx) => (
        <div key={message.id ?? idx}>
          {message.type}: {message.content}
        </div>
      ))}

      {stream.isLoading && <div>Loading...</div>}
      {stream.error && <div>Error: {stream.error.message}</div>}
    </div>
  );
}
```

LangChain Docs

LangChain Docs

Learn how to **deploy your agents to LangSmith** for production-ready hosting with built-in observability, authentication, and scaling.

`useStream` parameters

`assistantId` `string` `required`

The ID of the agent to connect to. When using LangSmith deployments, this must match the agent ID shown in your deployment dashboard. For custom API deployments or local development, this can be any string that your server uses to identify the agent.

`apiUrl` `string`

The URL of the LangGraph server. Defaults to `http://localhost:2024` for local development.

`threadId` `string`

Connect to an existing thread instead of creating a new one. Useful for resuming conversations.

`onThreadId` `(id: string) => void`

Callback invoked when a new thread is created. Use this to persist the thread ID for later use.

`reconnectOnMount` `boolean | (() => Storage)`

Automatically resume an ongoing run when the component mounts. Set to `true` to use session storage, or provide a custom storage function.

`onCreated` `(run: Run) => void`

Callback invoked when a new run is created. Useful for persisting run metadata for resumption.

`onError` `(error: Error) => void`

Callback invoked when an error occurs during streaming.

`onFinish` `(state: StateType, run?: Run) => void`

Callback invoked when the stream completes successfully with the final state.

`onCustomEvent` `(data: unknown, context: { mutate }) => void`

Handle custom events emitted from your agent using the `writer`. See **Custom streaming events**.

`onUpdateEvent` `(data: unknown, context: { mutate }) => void`

Handle metadata events with run and thread information.

`messagesKey` `string` `default:"messages"`

The key in the graph state that contains the messages array.

`throttle` `boolean` `default:"true"`

Batch state updates for better rendering performance. Disable for immediate updates.

`initialValues` `StateType | null`

Initial state values to display while the first stream is loading. Useful for showing cached thread data immediately.

`useStream` return values

`messages` `Message[]`

All messages in the current thread, including both human and AI messages.

`values` `StateType`

The current graph state values. Type is inferred from the agent or graph type parameter.

`isLoading` `boolean`

Whether a stream is currently in progress. Use this to show loading indicators.

`rror` `Error | null`

Current interrupt requiring user input, such as human-in-the-loop approval requests.

`toolCalls` `ToolCallWithResult[]`

All tool calls across all messages, with their results and state (`pending` , `completed` , or `error`).

`submit` `(input, options?) => Promise<void>`

Submit new input to the agent. Pass `null` as input when resuming from an interrupt with a command. Options include `checkpoint` for branching, `optimisticValues` for optimistic updates, and `threadId` for optimistic thread creation.

`stop` `() => void`

Stop the current stream immediately.

`joinStream` `(runId: string) => void`

Resume an existing stream by run ID. Use with `onCreated` for manual stream resumption.

`setBranch` `(branch: string) => void`

Switch to a different branch in the conversation history.

`getToolCalls` `(message) => ToolCall[]`

Get all tool calls for a specific AI message.

`experimental_branchTree` `BranchTree`

Tree representation of the thread for advanced branching controls in non-message based graphs.

Thread management

Keep track of conversations with built-in thread management. You can access the current thread ID and get notified when new threads are created:

```
import { useState } from "react";
import { useStream } from "@langchain/langgraph-sdk/react";

function Chat() {
  const [threadId, setThreadId] = useState<string | null>(null);

  const stream = useStream({
    apiUrl: "http://localhost:2024",
    assistantId: "agent",
    threadId: threadId,
    onThreadId: setThreadId,
  });

  // threadId is updated when a new thread is created
  // Store it in URL params or localStorage for persistence
}
```

ecommend storing the `threadId` to let users resume conversations after page refreshes.

refresh, ensuring no messages and events generated during the downtime are lost.

```
const stream = useStream({
  apiUrl: "http://localhost:2024",
  assistantId: "agent",
  reconnectOnMount: true,
});
```

By default the ID of the created run is stored in `window.sessionStorage` , which can be swapped by passing a custom storage function:

```
const stream = useStream({
  apiUrl: "http://localhost:2024",
  assistantId: "agent",
  reconnectOnMount: () => window.localStorage,
});
```

For manual control over the resumption process, use the run callbacks to persist metadata and `joinStream` to resume:

```
  const stream = useStream({
    apiUrl: "http://localhost:2024",
    assistantId: "agent",
    threadId,
    onCreated: (run) => {
      // Persist run ID when stream starts
      window.sessionStorage.setItem(`resume:${run.thread_id}`, run.run_id);
    },
    onFinish: (_, run) => {
      // Clean up when stream completes
      window.sessionStorage.removeItem(`resume:${run?.thread_id}`);
    },
  });

  // Resume stream on mount if there's a stored run ID
  const joinedThreadId = useRef<string | null>(null);
  useEffect(() => {
    if (!threadId) return;
    const runId = window.sessionStorage.getItem(`resume:${threadId}`);
    if (runId && joinedThreadId.current !== threadId) {
      stream.joinStream(runId);
      joinedThreadId.current = threadId;
    }
  }, [threadId]);

  const handleSubmit = (text: string) => {
    // Use streamResumable to ensure events aren't lost
    stream.submit(
      { messages: [{ type: "human", content: text }] },
      { streamResumable: true }
    );
  };
}
```

thread persistence in the `session-persistence` example.

Optimistic updates

You can optimistically update the client state before performing a network request, providing immediate feedback to the user:

```
const stream = useStream({
  apiUrl: "http://localhost:2024",
  assistantId: "agent",
});

const handleSubmit = (text: string) => {
  const newMessage = { type: "human" as const, content: text };

  stream.submit(
    { messages: [newMessage] },
    {
      optimisticValues(prev) {
        const prevMessages = prev.messages ?? [];
        return { ...prev, messages: [...prevMessages, newMessage] };
      },
    }
  );
};
```

Optimistic thread creation

Use the `threadId` option in `submit` to enable optimistic UI patterns where you need to the thread ID before the thread is created:

```
  const [threadId, setThreadId] = useState<string | null>(null);
  const [optimisticThreadId] = useState(() => crypto.randomUUID());

  const stream = useStream({
    apiUrl: "http://localhost:2024",
    assistantId: "agent",
    threadId,
    onThreadId: setThreadId,
  });

  const handleSubmit = (text: string) => {
    // Navigate immediately without waiting for thread creation
    window.history.pushState({}, "", `/threads/${optimisticThreadId}`);

    // Create thread with the predetermined ID
    stream.submit(
      { messages: [{ type: "human", content: text }] },
      { threadId: optimisticThreadId }
    );
  };
}
```

Cached thread display

Use the `initialValues` option to display cached thread data immediately while the history is being loaded from the server:

```
    threadId,
    initialValues: cachedData?.values,
  });

  // Shows cached messages instantly, then updates when server responds
}
```

Branching

Create alternate conversation paths by editing previous messages or regenerating AI responses. Use `getMessagesMetadata()` to access checkpoint information for branching:

```
function Chat() {
  const stream = useStream({
    apiUrl: "http://localhost:2024",
    assistantId: "agent",
  });

  return (
    <div>
      {stream.messages.map((message) => {
        const meta = stream.getMessagesMetadata(message);
        const parentCheckpoint = meta?.firstSeenState?.parent_checkpoint;

        return (
          <div key={message.id}>
            <div>{message.content as string}</div>

            {/* Edit human messages */}
            {message.type === "human" && (
              <button
                onClick={() => {
                  const newContent = prompt("Edit message:", message.content as
                  if (newContent) {
                    stream.submit(
                      { messages: [{ type: "human", content: newContent }] },
                      { checkpoint: parentCheckpoint }
                    );
                  }
                }}
              >
                Edit
              </button>
            )}

            {/* Regenerate AI messages */}
            {message.type === "ai" && (
              </button>
            )}

            {/* Switch between branches */}
            <BranchSwitcher
              branch={meta?.branch}
              branchOptions={meta?.branchOptions}
              onSelect={(branch) => stream.setBranch(branch)}
            />
          </div>
        );
      })}
    </div>
  );
}
```

For advanced use cases, use the `experimental_branchTree` property to get the tree representation of the thread for non-message based graphs.

Try the branching example

See a complete implementation of conversation branching with edit, regenerate, and branch switching in the `branching-chat` example.

Type-safe streaming

The `useStream` hook supports full type inference when used with agents created via `createAgent` or graphs created with `StateGraph`. Pass `typeof agent` or `typeof graph` as the type parameter to automatically infer tool call types.

agent.ts Chat.tsx

```ts
import { createAgent, tool } from "langchain";
import { z } from "zod";

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

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

With `StateGraph`

For custom `StateGraph` applications, the state types are inferred from the graph's annotation:

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

const workflow = new StateGraph(MessagesAnnotation)
  .addNode("agent", async (state) => {
    const response = await model.invoke(state.messages);
    return { messages: [response] };
  })
  .addEdge(START, "agent")
  .addEdge("agent", END);

export const graph = workflow.compile();
```

With Annotation types

If you’re using LangGraph.js, you can reuse your graph’s annotation types. Make sure to only import types to avoid importing the entire LangGraph.js runtime:

```
  type UpdateType,
} from "@langchain/langgraph/web";

const AgentState = Annotation.Root({
  ...MessagesAnnotation.spec,
  context: Annotation<string>(),
});

const stream = useStream<
  StateType<typeof AgentState.spec>,
  { UpdateType: UpdateType<typeof AgentState.spec> }
>({
  apiUrl: "http://localhost:2024",
  assistantId: "agent",
});
```

Advanced type configuration

You can specify additional type parameters for interrupts, custom events, and configurable options:

```
const stream = useStream<
  State,
  {
    UpdateType: { messages: Message[] | Message; context?: string };
    InterruptType: string;
    CustomEventType: { type: "progress" | "debug"; payload: unknown };
    ConfigurableType: { model: string };
  }
>({
  apiUrl: "http://localhost:2024",
  assistantId: "agent",
});

// stream.interrupt is typed as string | undefined
// onCustomEvent receives typed events
```

Rendering tool calls

Use `getToolCalls` to extract and render tool calls from AI messages. Tool calls include the call details, result (if completed), and state.

```
import { ToolCallCard } from "./ToolCallCard";
import { MessageBubble } from "./MessageBubble";

function Chat() {
  const stream = useStream<typeof agent>({
    assistantId: "agent",
    apiUrl: "http://localhost:2024",
  });

  return (
    <div className="flex flex-col gap-4">
      {stream.messages.map((message, idx) => {
        if (message.type === "ai") {
          const toolCalls = stream.getToolCalls(message);

          if (toolCalls.length > 0) {
            return (
              <div key={message.id ?? idx} className="flex flex-col gap-2">
                {toolCalls.map((toolCall) => (
                  <ToolCallCard key={toolCall.id} toolCall={toolCall} />
                ))}
              </div>
            );
          }
        }

        return <MessageBubble key={message.id ?? idx} message={message} />;
      })}
    </div>
  );
}
```

Custom streaming events

Stream custom data from your agent using the `writer` in your tools or nodes. Handle these events in the UI with the `onCustomEvent` callback.

```
// Define your custom event types
interface ProgressData {
  type: "progress";
  id: string;
  message: string;
  progress: number;
}

const analyzeDataTool = tool(
  async ({ dataSource }, config: ToolRuntime) => {
    const steps = ["Connecting...", "Fetching...", "Processing...", "Done!"];

    for (let i = 0; i < steps.length; i++) {
      // Emit progress events during execution
      config.writer?.({
        type: "progress",
        id: `analysis-${Date.now()}`,
        message: steps[i],
        progress: ((i + 1) / steps.length) * 100,
      } satisfies ProgressData);

      await new Promise((resolve) => setTimeout(resolve, 500));
    }

    return JSON.stringify({ result: "Analysis complete" });
  },
  {
    name: "analyze_data",
    description: "Analyze data with progress updates",
    schema: z.object({
      dataSource: z.string().describe("Data source to analyze"),
    }),
  }
```

badges, and file operation cards in the `custom-streaming` example.

Event handling

The `useStream` hook provides callback options that give you access to different types of streaming events. You don't need to explicitly configure stream modes—just pass callbacks for the event types you want to handle:

```
  // Handle state updates after each graph step
  onUpdateEvent: (update, options) => {
    console.log("Graph update:", update);
  },

  // Handle custom events streamed from your graph
  onCustomEvent: (event, options) => {
    console.log("Custom event:", event);
  },

  // Handle metadata events with run/thread info
  onMetadataEvent: (metadata) => {
    console.log("Run ID:", metadata.run_id);
    console.log("Thread ID:", metadata.thread_id);
  },

  onError: (error) => {
    console.error("Stream error:", error);
  },

  onFinish: (state, options) => {
    console.log("Stream finished with final state:", state);
  },
});
```

Available callbacks

`onCustomEvent`	Called when a custom event is received from your graph	`custom`
`onMetadataEvent`	Called with run and thread metadata	`metadata`
`onError`	Called when an error occurs	-
`onFinish`	Called when the stream completes	-

Multi-agent streaming

When working with multi-agent systems or graphs with multiple nodes, use message metadata to identify which node generated each message. This is particularly useful when multiple LLMs run in parallel and you want to display their outputs with distinct visual styling.

```
import { MessageBubble } from "./MessageBubble";

// Node configuration for visual display
const NODE_CONFIG: Record<string, { label: string; color: string }> = {
  researcher_analytical: { label: "Analytical Research", color: "cyan" },
  researcher_creative: { label: "Creative Research", color: "purple" },
  researcher_practical: { label: "Practical Research", color: "emerald" },
};

function MultiAgentChat() {
  const stream = useStream<typeof agent>({
    assistantId: "parallel-research",
    apiUrl: "http://localhost:2024",
  });

  return (
    <div className="flex flex-col gap-4">
      {stream.messages.map((message, idx) => {
        if (message.type !== "ai") {
          return <MessageBubble key={message.id ?? idx} message={message} />;
        }

        // Get streaming metadata to identify the source node
        const metadata = stream.getMessagesMetadata?.(message);
        const nodeName =
          (metadata?.streamMetadata?.langgraph_node as string) ||
          (message as { name?: string }).name;

        const config = nodeName ? NODE_CONFIG[nodeName] : null;

        if (!config) {
          return <MessageBubble key={message.id ?? idx} message={message} />;
        }

        return (
          <div
```

```
              {config.label}
            </div>
            <div className="text-neutral-200 whitespace-pre-wrap">
              {typeof message.content === "string" ? message.content : ""}
            </div>
          </div>
        );
      })}
    </div>
  );
}
```

↗

Try the parallel research example

See a complete implementation of multi-agent streaming with three parallel researchers and distinct visual styling in the `parallel-research` example.

Human-in-the-loop

Handle interrupts when the agent requires human approval for tool execution. Learn more in the **How to handle interrupts** guide.

```
import type { HITLRequest, HITLResponse } from "langchain";
import type { agent } from "./agent";
import { MessageBubble } from "./MessageBubble";

function HumanInTheLoopChat() {
  const stream = useStream<typeof agent, { InterruptType: HITLRequest }>({
    assistantId: "human-in-the-loop",
    apiUrl: "http://localhost:2024",
  });

  const [isProcessing, setIsProcessing] = useState(false);

  // Type assertion for interrupt value
  const hitlRequest = stream.interrupt?.value as HITLRequest | undefined;

  const handleApprove = async (index: number) => {
    if (!hitlRequest) return;
    setIsProcessing(true);

    try {
      const decisions: HITLResponse["decisions"] =
        hitlRequest.actionRequests.map((_, i) =>
          i === index ? { type: "approve" } : { type: "approve" }
        );

      await stream.submit(null, {
        command: {
          resume: { decisions } as HITLResponse,
        },
      });
    } finally {
      setIsProcessing(false);
    }
  };

  const handleReject = async (index: number, reason: string) => {
      const decisions: HITLResponse["decisions"] =
        hitlRequest.actionRequests.map((_, i) =>
          i === index
            ? { type: "reject", message: reason }
            : { type: "reject", message: "Rejected along with other actions" }
        );

      await stream.submit(null, {
        command: {
          resume: { decisions } as HITLResponse,
        },
      });
    } finally {
      setIsProcessing(false);
    }
  };

  return (
    <div>
      {/* Render messages */}
      {stream.messages.map((message, idx) => (
        <MessageBubble key={message.id ?? idx} message={message} />
      ))}

      {/* Render approval UI when interrupted */}
      {hitlRequest && hitlRequest.actionRequests.length > 0 && (
        <div className="bg-amber-900/20 border border-amber-500/30 rounded-xl p
          <h3 className="text-amber-400 font-semibold mb-4">
            Action requires approval
          </h3>

          {hitlRequest.actionRequests.map((action, idx) => (
            <div
              key={idx}
              className="bg-neutral-900 rounded-lg p-4 mb-4 last:mb-0"
            >
```

```
              </div>

              <pre className="text-xs bg-black rounded p-2 mb-3 overflow-x-aut
                {JSON.stringify(action.args, null, 2)}
              </pre>

              <div className="flex gap-2">
                <button
                  onClick={() => handleApprove(idx)}
                  disabled={isProcessing}
                  className="px-3 py-1.5 bg-green-600 hover:bg-green-700 text-w
                >
                  Approve
                </button>
                <button
                  onClick={() => handleReject(idx, "User rejected")}
                  disabled={isProcessing}
                  className="px-3 py-1.5 bg-red-600 hover:bg-red-700 text-white
                >
                  Reject
                </button>
              </div>
            </div>
          ))}
        </div>
      )}
    </div>
  );
}
```

↗

Try the human-in-the-loop example

ee a complete implementation of approval workflows with approve, reject, and dit actions in the `human-in-the-loop` example.

are developed.

When using models with extended reasoning capabilities (like OpenAI's reasoning models or Anthropic's extended thinking), the thinking process is embedded in the message content. You'll need to extract and display it separately.

```
import type { agent } from "./agent";
import { getReasoningFromMessage, getTextContent } from "./utils";

function ReasoningChat() {
  const stream = useStream<typeof agent>({
    assistantId: "reasoning-agent",
    apiUrl: "http://localhost:2024",
  });

  return (
    <div className="flex flex-col gap-4">
      {stream.messages.map((message, idx) => {
        if (message.type === "ai") {
          const reasoning = getReasoningFromMessage(message);
          const textContent = getTextContent(message);

          return (
            <div key={message.id ?? idx}>
              {/* Render reasoning bubble if present */}
              {reasoning && (
                <div className="mb-4">
                  <div className="text-xs font-medium text-amber-400/80 mb-2">
                    Reasoning
                  </div>
                  <div className="bg-amber-950/50 border border-amber-500/20 r
                    <div className="text-sm text-amber-100/90 whitespace-pre-w
                      {reasoning}
                    </div>
                  </div>
                </div>
              )}

              {/* Render text content */}
              {textContent && (
                <div className="text-neutral-100 whitespace-pre-wrap">
                  {textContent}
        }

        return <MessageBubble key={message.id ?? idx} message={message} />;
      })}

      {stream.isLoading && (
        <div className="flex items-center gap-2 text-amber-400/70">
          <span className="text-sm">Thinking...</span>
        </div>
      )}
    </div>
  );
}
```

Try the reasoning example

See a complete implementation of reasoning token display with OpenAI and Anthropic models in the `reasoning-agent` example.

Custom state types

For custom LangGraph applications, embed your tool call types in your state's messages property.

```
type MyToolCalls =
  | { name: "search"; args: { query: string }; id?: string }
  | { name: "calculate"; args: { expression: string }; id?: string };

// Embed tool call types in your state's messages
interface MyGraphState {
  messages: Message<MyToolCalls>[];
  context?: string;
}

function CustomGraphChat() {
  const stream = useStream<MyGraphState>({
    assistantId: "my-graph",
    apiUrl: "http://localhost:2024",
  });

  // stream.values is typed as MyGraphState
  // stream.toolCalls[0].call.name is typed as "search" | "calculate"
}
```

You can also specify additional type configuration for interrupts and configurable options:

```
function CustomGraphChat() {
  const stream = useStream<
    MyGraphState,
    {
      InterruptType: { question: string };
      ConfigurableType: { userId: string };
    }
  >({
    assistantId: "my-graph",
    apiUrl: "http://localhost:2024",
  });

  // stream.interrupt is typed as { question: string } | undefined
}
```

Custom transport

For custom API endpoints or non-standard deployments, use the `transport` option with `FetchStreamTransport` to connect to any streaming API.

```
// Create transport with custom request handling
const transport = useMemo(() => {
  return new FetchStreamTransport({
    apiUrl: "/api/my-agent",
    onRequest: async (url: string, init: RequestInit) => {
      // Inject API key or other custom data into requests
      const customBody = JSON.stringify({
        ...(JSON.parse(init.body as string) || {}),
        apiKey,
      });

      return {
        ...init,
        body: customBody,
        headers: {
          ...init.headers,
          "X-Custom-Header": "value",
        },
      };
    },
  });
}, [apiKey]);

const stream = useStream({
  transport,
});

// Use stream as normal
return (
  <div>
    {stream.messages.map((message, idx) => (
      <MessageBubble key={message.id ?? idx} message={message} />
    ))}
  </div>
);
```

Streaming overview — Server-side streaming with LangChain agents

useStream API Reference — Full API documentation

Agent Chat UI — Pre-built chat interface for LangGraph agents

Human-in-the-loop — Configuring interrupts for human review

Multi-agent systems — Building agents with multiple LLMs

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 Academy

Blog

Trust Center

Powered by mintlify

LangChain Docs