LangChain Docs

Core components › Short-term memory

Core components

Short-term memory

Copy page

Overview

Memory is a system that remembers information about previous interactions. For AI agents, memory is crucial because it lets them remember previous interactions, learn from feedback, and adapt to user preferences. As agents tackle more complex tasks with numerous user interactions, this capability becomes essential for both efficiency and user satisfaction.

Short term memory lets your application remember previous interactions within a single thread or conversation.

> A thread organizes multiple interactions in a session, similar to the way email groups messages in a single conversation.

Conversation history is the most common form of short-term memory. Long conversations pose a challenge to today's LLMs; a full history may not fit inside an LLM's context window, resulting in an context loss or errors.

Even if your model supports the full context length, most LLMs still perform poorly over long contexts. They get "distracted" by stale or off-topic content, all while suffering from slower response times and higher costs.

Chat models accept context using **messages**, which include instructions (a system message) and inputs (human messages). In chat applications, messages alternate between human inputs and model responses, resulting in a list of messages that grows longer over time. Because context windows are limited, many applications can benefit

LangChain Docs

Core components › **Short-term memory**

To add short-term memory (thread-level persistence) to an agent, you need to specify a `checkpointer` when creating an agent.

> LangChain's agent manages short-term memory as a part of your agent's state.
>
> By storing these in the graph's state, the agent can access the full context for a given conversation while maintaining separation between different threads.
>
> State is persisted to a database (or memory) using a checkpointer so the thread can be resumed at any time.
>
> Short-term memory updates when the agent is invoked or a step (like a tool call) is completed, and the state is read at the start of each step.

```
import { createAgent } from "langchain";
import { MemorySaver } from "@langchain/langgraph";

const checkpointer = new MemorySaver();

const agent = createAgent({
    model: "claude-sonnet-4-5-20250929",
    tools: [],
    checkpointer,
});

await agent.invoke(
    { messages: [{ role: "user", content: "hi! i am Bob" }] },
    { configurable: { thread_id: "1" } }
);
```

In production

In production, use a checkpointer backed by a database:

LangChain Docs

Core components › **Short-term memory**

LangChain Docs

Core components › **Short-term memory**

LangChain Docs

Core components › **Short-term memory**

LangChain Docs

Core components › **Short-term memory**

LangChain Docs

Core components › **Short-term memory**

LangChain Docs

Core components › **Short-term memory**

LangChain Docs

Core components › **Short-term memory**

LangChain Docs

Core components › **Short-term memory**

LangChain Docs

Core components › **Short-term memory**

LangChain Docs

Core components › **Short-term memory**

LangChain Docs

Core components › **Short-term memory**

LangChain Docs

Core components › **Short-term memory**

LangChain Docs

Core components › **Short-term memory**

```
import { PostgresSaver } from "@langchain/langgraph-checkpoint-post

const DB_URI = "postgresql://postgres:postgres@localhost:5442/postgres?sslmode=
const checkpointer = PostgresSaver.fromConnString(DB_URI);
```

Customizing agent memory

You can extend the agent state by creating custom middleware with a state schema. Custom state schemas can be passed using the `stateSchema` parameter in middleware.

```
const customStateSchema = z.object({
    userId: z.string(),
    preferences: z.record(z.string(), z.any()),
});

const stateExtensionMiddleware = createMiddleware({
    name: "StateExtension",
    stateSchema: customStateSchema,
});

const checkpointer = new MemorySaver();
const agent = createAgent({
    model: "gpt-5",
    tools: [],
    middleware: [stateExtensionMiddleware],
    checkpointer,
});

// Custom state can be passed in invoke
const result = await agent.invoke({
    messages: [{ role: "user", content: "Hello" }],
    userId: "user_123",
    preferences: { theme: "dark" },
});
```

Common patterns

With **short-term memory** enabled, long conversations can exceed the LLM's context window. Common solutions are:

(before calling LLM)

state permanently

Summarize messages

Summarize earlier messages in the history and replace them with a summary

Custom strategies

Custom strategies (e.g., message filtering, etc.)

This allows the agent to keep track of the conversation without exceeding the LLM's context window.

Trim messages

Most LLMs have a maximum supported context window (denominated in tokens).

One way to decide when to truncate messages is to count the tokens in the message history and truncate whenever it approaches that limit. If you're using LangChain, you can use the trim messages utility and specify the number of tokens to keep from the list, as well as the `strategy` (e.g., keep the last `maxTokens`) to use for handling the boundary.

To trim message history in an agent, use `createMiddleware` with a `beforeModel` hook:

LangChain Docs

Core components › Short-term memory

```
const trimMessages = createMiddleware({
  name: "TrimMessages",
  beforeModel: (state) => {
    const messages = state.messages;

    if (messages.length <= 3) {
      return; // No changes needed
    }

    const firstMsg = messages[0];
    const recentMessages =
      messages.length % 2 === 0 ? messages.slice(-3) : messages.slice(-4);
    const newMessages = [firstMsg, ...recentMessages];

    return {
      messages: [
        new RemoveMessage({ id: REMOVE_ALL_MESSAGES }),
        ...newMessages,
      ],
    };
  },
});

const checkpointer = new MemorySaver();
const agent = createAgent({
  model: "gpt-4o",
  tools: [],
  middleware: [trimMessages],
  checkpointer,
});
```

Delete messages

LangChain Docs

Core components › **Short-term memory**

To delete messages from the graph state, you can use the `RemoveMessage`. For `RemoveMessage` to work, you need to use a state key with `messagesStateReducer` reducer, like `MessagesValue`.

To remove specific messages:

```
import { RemoveMessage } from "@langchain/core/messages";

const deleteMessages = (state) => {
    const messages = state.messages;
    if (messages.length > 2) {
        // remove the earliest two messages
        return {
        messages: messages
            .slice(0, 2)
            .map((m) => new RemoveMessage({ id: m.id })),
        };
    }
};
```

⚠ When deleting messages, **make sure** that the resulting message history is valid. Check the limitations of the LLM provider you're using. For example:

- Some providers expect message history to start with a `user` message
- Most providers require `assistant` messages with tool calls to be followed by corresponding `tool` result messages.

LangChain Docs

Core components › **Short-term memory**

```
const deleteOldMessages = createMiddleware({
  name: "DeleteOldMessages",
  afterModel: (state) => {
    const messages = state.messages;
    if (messages.length > 2) {
      // remove the earliest two messages
      return {
        messages: messages
          .slice(0, 2)
          .map((m) => new RemoveMessage({ id: m.id! })),
      };
    }
    return;
  },
});

const agent = createAgent({
  model: "gpt-4o",
  tools: [],
  systemPrompt: "Please be concise and to the point.",
  middleware: [deleteOldMessages],
  checkpointer: new MemorySaver(),
});

const config = { configurable: { thread_id: "1" } };

const streamA = await agent.stream(
  { messages: [{ role: "user", content: "hi! I'm bob" }] },
  { ...config, streamMode: "values" }
);
for await (const event of streamA) {
  const messageDetails = event.messages.map((message) => [
    message.getType(),
    message.content,
  ]);
```

```
    messages: [{ role: "user", content: "what's my name?" }],
  },
  { ...config, streamMode: "values" }
);
for await (const event of streamB) {
  const messageDetails = event.messages.map((message) => [
    message.getType(),
    message.content,
  ]);
  console.log(messageDetails);
}
```

```
[[ "human", "hi! I'm bob" ]]
[[ "human", "hi! I'm bob" ], [ "ai", "Hello, Bob! How can I assist you today?"
[[ "human", "hi! I'm bob" ], [ "ai", "Hello, Bob! How can I assist you today?"
[[ "human", "hi! I'm bob" ], [ "ai", "Hello, Bob! How can I assist you today" ]
[[ "human", "hi! I'm bob" ], [ "ai", "Hello, Bob! How can I assist you today?"
[[ "human", "what's my name?" ], [ "ai", "Your name is Bob, as you mentioned. H
```

Summarize messages

The problem with trimming or removing messages, as shown above, is that you may lose information from culling of the message queue. Because of this, some applications benefit from a more sophisticated approach of summarizing the message history using a chat model.

LangChain Docs

Core components › **Short-term memory**

To summarize message history in an agent, use the built-in `summarizationMiddleware`:

```
import { createAgent, summarizationMiddleware } from "langchain";
import { MemorySaver } from "@langchain/langgraph";

const checkpointer = new MemorySaver();

const agent = createAgent({
  model: "gpt-4o",
  tools: [],
  middleware: [
    summarizationMiddleware({
      model: "gpt-4o-mini",
      trigger: { tokens: 4000 },
      keep: { messages: 20 },
    }),
  ],
  checkpointer,
});

const config = { configurable: { thread_id: "1" } };
await agent.invoke({ messages: "hi, my name is bob" }, config);
await agent.invoke({ messages: "write a short poem about cats" }, config);
await agent.invoke({ messages: "now do the same but for dogs" }, config);
const finalResponse = await agent.invoke({ messages: "what's my name?" }, confi

console.log(finalResponse.messages.at(-1)?.content);
// Your name is Bob!
```

See `summarizationMiddleware` for more configuration options.

Tools

Read short-term memory in a tool

Access short term memory (state) in a tool using the `runtime` parameter (typed as `ToolRuntime`).

The `runtime` parameter is hidden from the tool signature (so the model doesn't see it), but the tool can access the state through it.

LangChain Docs

Core components › **Short-term memory**

```
  userId: z.string(),
});

const getUserInfo = tool(
  async (_, config: ToolRuntime<z.infer<typeof stateSchema>>) => {
    const userId = config.state.userId;
    return userId === "user_123" ? "John Doe" : "Unknown User";
  },
  {
    name: "get_user_info",
    description: "Get user info",
    schema: z.object({}),
  }
);

const agent = createAgent({
  model: "gpt-5-nano",
  tools: [getUserInfo],
  stateSchema,
});

const result = await agent.invoke(
  {
    messages: [{ role: "user", content: "what's my name?" }],
    userId: "user_123",
  },
  {
    context: {},
  }
);

console.log(result.messages.at(-1)?.content);
// Outputs: "Your name is John Doe."
```

LangChain Docs

Core components › **Short-term memory**

This is useful for persisting intermediate results or making information accessible to subsequent tools or prompts.

LangChain Docs

Core components › Short-term memory

```
const CustomState = z.object({
  userId: z.string().optional(),
});

const updateUserInfo = tool(
  async (_, config: ToolRuntime<typeof CustomState>) => {
    const userId = config.state.userId;
    const name = userId === "user_123" ? "John Smith" : "Unknown user";
    return new Command({
      update: {
        userName: name,
        // update the message history
        messages: [
          new ToolMessage({
            content: "Successfully looked up user information",
            tool_call_id: config.toolCall?.id ?? "",
          }),
        ],
      },
    });
  },
  {
    name: "update_user_info",
    description: "Look up and update user info.",
    schema: z.object({}),
  }
);

const greet = tool(
  async (_, config) => {
    const userName = config.context?.userName;
    return `Hello ${userName}!`;
  },
  {
    name: "greet",
```

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

const result = await agent.invoke({
  messages: [{ role: "user", content: "greet the user" }],
  userId: "user_123",
});

console.log(result.messages.at(-1)?.content);
// Output: "Hello! I'm here to help — what would you like to do today?"
```

Prompt

Access short term memory (state) in middleware to create dynamic prompts based on conversation history or custom state fields.

LangChain Docs

Core components › Short-term memory

```
  userName: z.string(),
});
type ContextSchema = z.infer<typeof contextSchema>;

const getWeather = tool(
  async ({ city }) => {
    return `The weather in ${city} is always sunny!`;
  },
  {
    name: "get_weather",
    description: "Get user info",
    schema: z.object({
      city: z.string(),
    }),
  }
);

const agent = createAgent({
  model: "gpt-5-nano",
  tools: [getWeather],
  contextSchema,
  middleware: [
    dynamicSystemPromptMiddleware<ContextSchema>((_, config) => {
      return `You are a helpful assistant. Address the user as ${config.context
    }),
  ],
});

const result = await agent.invoke(
  {
    messages: [{ role: "user", content: "What is the weather in SF?" }],
  },
  {
    context: {
      userName: "John Smith",
```

LangChain Docs

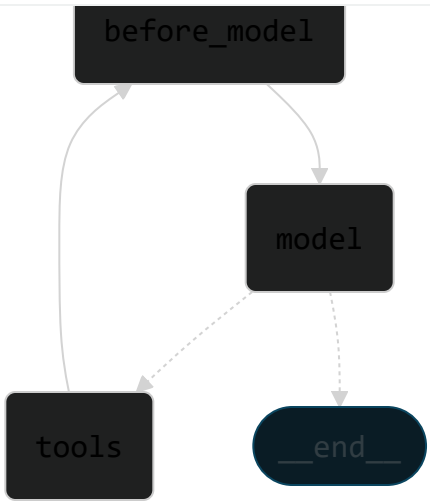
Core components › Short-term memory

```
  console.log(message);
}
/**
 * HumanMessage {
 *   "content": "What is the weather in SF?",
 *   // ...
 * }
 * AIMessage {
 *   // ...
 *   "tool_calls": [
 *     {
 *       "name": "get_weather",
 *       "args": {
 *         "city": "San Francisco"
 *       },
 *       "type": "tool_call",
 *       "id": "call_tCidbv0apTpQpEWb3O2zQ4Yx"
 *     }
 *   ],
 *   // ...
 * }
 * ToolMessage {
 *   "content": "The weather in San Francisco is always sunny!",
 *   "tool_call_id": "call_tCidbv0apTpQpEWb3O2zQ4Yx"
 *   // ...
 * }
 * AIMessage {
 *   "content": "John Smith, here's the latest: The weather in San Francisco is
 *   // ...
 * }
 */
```

Store model

LangChain Docs

Core components › **Short-term memory**



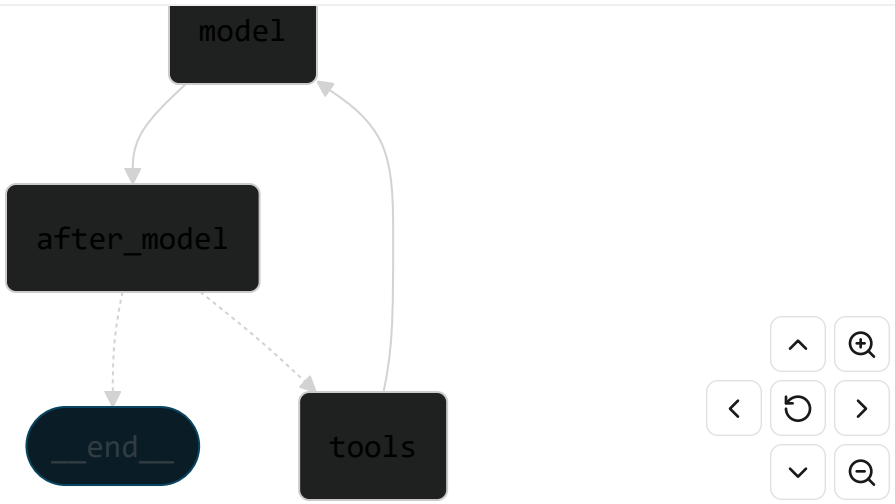
```
const trimMessageHistory = createMiddleware({
  name: "TrimMessages",
  beforeModel: async (state) => {
    const trimmed = await trimMessages(state.messages, {
      maxTokens: 384,
      strategy: "last",
      startOn: "human",
      endOn: ["human", "tool"],
      tokenCounter: (msgs) => msgs.length,
    });
    return {
      messages: [new RemoveMessage({ id: REMOVE_ALL_MESSAGES }), ...trimmed],
    };
  },
});

const checkpointer = new MemorySaver();
const agent = createAgent({
  model: "gpt-5-nano",
  tools: [],
  middleware: [trimMessageHistory],
  checkpointer,
});
```

After model

LangChain Docs

Core components › **Short-term memory**



```
const validateResponse = createMiddleware({
  name: "ValidateResponse",
  afterModel: (state) => {
    const lastMessage = state.messages.at(-1)?.content;
    if (
      typeof lastMessage === "string" &&
      lastMessage.toLowerCase().includes("confidential")
    ) {
      return {
        messages: [
          new RemoveMessage({ id: REMOVE_ALL_MESSAGES }),
          ...state.messages,
        ],
      };
    }
    return;
  },
});

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

LangChain Docs

Core components › **Short-term memory**

LangChain Docs

Resources

Forum

Changelog

LangChain Academy

Trust Center

Company

About

Careers

Blog

Powered by mintlify

LangChain Docs

Core components › **Short-term memory**