Agent Memory

- Get started
- Agent Memory
- HTTP API
- API
- Workers API
- How Agent Memory works
- Concepts
- Namespaces and profiles
- Platform
- Limits
- Pricing

Get started

import { LinkCard, PackageManagers, Render, TypeScriptExample, WranglerConfig, } from "~/components";

Add Agent Memory to an agent so it can recall durable context across conversations.

This guide uses the Agents SDK to expose memory recall as a model-callable tool. The same pattern applies if you use another agent framework: store memories with `ingest()` or `remember()`, expose `recall()` through one of your agent's tools, and use the system prompt to tell the model when to search memory.

Prerequisites

You also need access to Agent Memory.

How agent memory works

Use `recall()` when the model needs relevant memory to answer or act. Use `ingest()` when you have conversation messages and want Agent Memory to extract durable memories automatically. Use `remember()` when your agent already knows the exact memory to store.

Do not call `ingest()` after every model turn. Instead, batch ingestion after the user goes idle, when a conversation is compacted, or at another natural checkpoint.

1. Create a project

Create a Worker project:

<Render file="c3-post-run-steps-hello-world" product="workers" params={{ type: "Worker only", lang: "TypeScript", }} />

Move into the project directory:

```
cd memory-agent
```

Install the dependencies used by this guide:

2. Create a namespace

A namespace scopes the memory profiles for your application. Create one with Wrangler:

You will use the namespace name, `my-agent`, in your Worker binding.

3. Configure bindings

Add an `agent_memory` binding to your Wrangler configuration, and register your agent's Durable Object.

```
name = "memory-agent"
main = "src/server.ts"
upload_source_maps = true
compatibility_date = "$today"
compatibility_flags = ["nodejs_compat"]

[ai]
binding = "AI"

[[agent_memory]]
binding = "MEMORY"
namespace = "my-agent"

[[durable_objects.bindings]]
name = "ChatAgent"
class_name = "ChatAgent"

[[migrations]]
tag = "v1"
new_sqlite_classes = ["ChatAgent"]
```

Generate local TypeScript types for your bindings:

4. Add memory recall as a tool

The model cannot use memory just because your application has a memory binding. You need to expose recall through a tool and instruct the model when to call it.

This guide builds the agent on a Durable Object with the Agents SDK Lifecycle. The `Sessions` capability stores the conversation. `agents/sessions` is experimental, so its API may change between releases.

Create `src/server.ts`. It defines the system prompt and a `search_memory` tool that calls `recall()`:

```
import { DurableObject } from "cloudflare:workers";
import { tool } from "ai";
import { z } from "zod";
import { Lifecycle } from "agents/lifecycle";
import { routeAgentRequest } from "agents/routing";
import { Sessions } from "agents/sessions";

const SYSTEM_PROMPT = `You are a helpful assistant.

Long-term memory is available through the search_memory tool.

MEMORY POLICY
- Search memory with search_memory when the user asks what you know or remember about them.
- Search memory when the request depends on prior sessions, preferences, project state,
conventions, decisions, or long-running tasks.
- Phrase memory searches as concise topics, not questions.
- Do not search memory to repeat something the user just said in the current conversation.
- When search_memory returns results, always incorporate them into your response. The results
are real memories from previous conversations.
- Treat recalled memories as helpful context, not guaranteed truth. If a memory is important
for an irreversible action, confirm with the user.`;

const MEMORY_PROFILE_NAME = "demo-user";

export class ChatAgent extends DurableObject<Env> {
  readonly sessions = new Sessions();
  readonly lifecycle = Lifecycle.install(this).use(this.sessions);
  readonly session = this.sessions.session();
```

```
function memoryTools() {
  return {
    search_memory: tool({
      description:
        "Search durable memory: facts, events, instructions, and tasks from prior conversations.",
      inputSchema: z.object({
        query: z.string().describe("A concise topic to search for"),
      }),
      execute: async ({ query }) => {
        const profile = await this.env.MEMORY.getProfile(MEMORY_PROFILE_NAME);
        const { answer } = await profile.recall(query, {
          responseLength: "short",
        });
        return answer || "No relevant memories found.";
      },
    }),
  };
}

export default {
  async fetch(request: Request, env: Env) {
    return (
      (await routeAgentRequest(request, env)) ??
      new Response("Not found", { status: 404 })
    );
  },
} satisfies ExportedHandler<Env>;
```

The system prompt is as important as the tool. It tells the model when to call `search_memory`, when not to call it, and how to treat recalled memory. The prompt is the same on every turn, so the model provider can keep it in its prompt cache.

5. Extract memories from conversation

Next, give your agent a way to add durable memories. In a chat agent, the usual path is to store the conversation in a session, then call `ingest()` after the user goes idle. The `Scheduler` capability runs the ingest later, even if the Durable Object is evicted in the meantime.

Replace the imports at the top of `src/server.ts`:

```
import { DurableObject } from "cloudflare:workers";
import {
  convertToModelMessages,
  generateText,
  stepCountIs,
  tool,
  type UIMessage,
} from "ai";
import { z } from "zod";
import { Lifecycle } from "agents/lifecycle";
import { createAI } from "agents/models/ai-sdk";
import { routeAgentRequest } from "agents/routing";
import { Scheduler } from "agents/schedules";
import { Sessions } from "agents/sessions";
```

Add the ingestion delay near the top of the file, after the imports:

```
const MEMORY_INGEST_DELAY_SECONDS = 10;
```

Then update `ChatAgent` with the following shape. The comment marks where to keep `memoryTools()` from step 4.

```
export class ChatAgent extends DurableObject<Env> {
  readonly sessions = new Sessions();
  readonly scheduler = new Scheduler({
    callbacks: {
      ingestMemory: () => this.runIngest(),
    },
  });
  readonly lifecycle = Lifecycle.install(this)
    .use(this.sessions)
    .use(this.scheduler);
  readonly session = this.sessions.session();

  // Keep `memoryTools()` from step 4 here.
```

```
async chat(message: string): Promise<string> {
  // RPC calls do not start the Lifecycle on their own.
  await this.lifecycle.start();

  await this.session.appendMessage({
    id: `user-${crypto.randomUUID()}`,
    role: "user",
    parts: [{ type: "text", text: message }],
  });

  await this.scheduleIngest();

  const ai = createAI({ binding: this.env.AI });
  const result = await generateText({
    model: ai("@cf/zai-org/glm-5.3-flash"),
    system: SYSTEM_PROMPT,
    messages: await convertToModelMessages(
      (await this.session.getHistory()) as UIMessage[],
    ),
    tools: this.memoryTools(),
    stopWhen: stepCountIs(5),
  });

  await this.session.appendMessage({
    id: `assistant-${crypto.randomUUID()}`,
    role: "assistant",
    parts: [{ type: "text", text: result.text }],
  });

  return result.text;
}

private async scheduleIngest() {
  // Replace any pending ingest, so it runs once the user goes idle.
  for (const schedule of await this.scheduler.list()) {
    if (schedule.callback === "ingestMemory") {
      await this.scheduler.cancel(schedule.id);
    }
  }
```

```
  await this.scheduler.set(MEMORY_INGEST_DELAY_SECONDS, "ingestMemory");
 }

 private async runIngest(): Promise<{ ingested: number }> {
  const cursor = (await this.ctx.storage.get<number>("ingestCursor")) ?? 0;
  const history = await this.session.getHistory();
  const messages = history
   .slice(cursor)
   .filter(
    (message) => message.role === "user" || message.role === "assistant",
   )
   .map((message) => ({
    role: message.role as "user" | "assistant",
    content: message.parts
     .map((part) =>
      part.type === "text" && typeof part.text === "string"
       ? part.text
       : "",
     )
     .filter(Boolean)
     .join("\n\n"),
   }))
   .filter((message) => message.content);

  if (messages.length > 0) {
   const profile = await this.env.MEMORY.getProfile(MEMORY_PROFILE_NAME);
   await profile.ingest(messages, { sessionId: this.lifecycle.name });
  }

  await this.ctx.storage.put("ingestCursor", history.length);
  return { ingested: messages.length };
 }
}
```

Replace the default export with a small test endpoint. Each `conversationId` maps to a separate Durable Object with its own session history. The Worker calls `chat()` over RPC:

```
export default {
 async fetch(request: Request, env: Env) {
  const url = new URL(request.url);
```

```
  if (request.method === "POST" && url.pathname === "/chat") {
    const { message, conversationId = "default" } =
      (await request.json()) as {
        message?: string;
        conversationId?: string;
      };
    if (!message) {
      return Response.json({ error: "Missing message" }, { status: 400 });
    }

    const agent = env.ChatAgent.getByName(conversationId);
    const response = await agent.chat(message);
    return Response.json({ response });
  }

  return (
    (await routeAgentRequest(request, env)) ??
    new Response("Not found", { status: 404 })
  );
 },
} satisfies ExportedHandler<Env>;
```

The ingestion path is `scheduleIngest()` and `runIngest()`. Each user message cancels the previous pending ingest and schedules a new one, so Agent Memory processes the conversation after the user goes idle instead of after every agent turn.

`runIngest()` keeps a cursor in the Durable Object's storage, so each batch only includes messages that have not already been ingested. The `sessionId` groups memories by conversation inside the shared memory profile.

This demo uses one Agent Memory profile, `demo-user`, across multiple conversations. In production, choose profile names that match your application scope, such as users, teams, tenants, or organizations.

You can also run the ingestion logic when a session is compacted. The important constraint is to ingest in batches at natural checkpoints, not after every agent turn. In production, choose an ingest delay that matches your application's user experience.

6. (Optional) Store explicit memories when needed

Automatic ingestion is enough for most apps. If you want the model to store a specific memory immediately, add a server-side tool whose execute function calls `remember()`.

Use this when the agent already knows the exact memory to store. For example, the model might call a `rememberMemory` tool after the user says: "Remember that I prefer concise answers."

If the model can call a memory-write tool, add system prompt instructions that define what is worth remembering and when to ask for confirmation. For many agents, automatic conversation ingestion is simpler and safer than giving the model a direct memory-write tool.

7. Test the app

Start local development:

Ask the first conversation to remember a durable preference:

```
curl -X POST "http://localhost:8787/chat" \
  -H "Content-Type: application/json" \
  -d '{"conversationId":"first-chat","message":"I prefer TypeScript examples and concise answers."}'
```

Wait at least 30 seconds before sending the next request. The code schedules ingestion to run 10 seconds after the user goes idle, and Agent Memory then needs additional time to extract, classify, and index the memories before they are available for recall.

Ask a different conversation a question that depends on durable memory. This request uses a different session history but the same Agent Memory profile.

```
curl -X POST "http://localhost:8787/chat" \
  -H "Content-Type: application/json" \
  -d '{"conversationId":"second-chat","message":"What do you know or remember about me and my preferences?"}'
```

The model should call `search_memory`, receive recalled memory from Agent Memory, and use that context in its response. The second conversation has no shared session history with the first, so

any knowledge of user preferences comes from Agent Memory.

Next steps

Agent Memory

import { CardGrid, Description, Feature, LinkButton, LinkTitleCard, RelatedProduct, } from "~/components";

Persistent, scoped memory for agents that need to remember users, organizations, and domain-specific context across conversations.

:::note

Agent Memory is in private beta. To understand the memory model, refer to How Agent Memory works.

:::

Agent Memory gives agents durable memory across conversations without requiring you to build your own extraction, storage, search, and summarization pipeline. Ingest conversations to extract memories automatically, or explicitly remember information when your application already knows what to store.

Use Agent Memory to remember user preferences, company operating rules, support history, project state, or the state of a specific object.

Agent Memory gives you:

- Isolated profiles for users, agents, tenants, teams, or application entities
- Namespaces for separating applications, environments, or memory layers
- Automatic extraction of facts, events, instructions, and tasks from conversations
- APIs to add, list, recall, and delete memories
- Profile summaries for observing what Agent Memory remembers

Get started <LinkButton variant="secondary" href="/agent-memory/concepts/how-agent-memory-works/"

> “

```
Learn how it works
```

Features

```
<Feature
header="Scoped memory profiles"
href="/agent-memory/concepts/namespaces-profiles/"
cta="Learn about profiles"
>
```

Store memories in isolated profiles for users, agents, teams, tenants, or application entities. Use namespaces to separate applications, environments, or memory layers.

<Feature header="Automatic memory extraction" href="/agent-memory/api/workers-api/#ingest-conversations" cta="Review the API"

> “

Ingest conversations to extract memories automatically, or explicitly add important information when your agent already knows what should be remembered.

<Feature header="Recall memories across agent executions" href="/agent-memory/api/workers-api/#recall-memories" cta="Recall memories"

> “

Retrieve relevant memory when an agent needs durable context that is not present in the current prompt, conversation, or execution.

Related products

Agent Memory is accessed through a Worker binding. Build serverless applications that use persistent memory.

Build AI-powered agents with persistent state, tool use, and real-time communication.

More resources

<LinkTitleCard title="Get started" href="/agent-memory/get-started/" icon="open-book"

> “

```
Add durable memory recall and ingestion to an agent.
```

<LinkTitleCard title="How Agent Memory works" href="/agent-memory/concepts/how-agent-memory-works/" icon="document"

> “

Learn how Agent Memory extracts, classifies, and retrieves knowledge from conversations.

<LinkTitleCard title="Workers API" href="/agent-memory/api/workers-api/" icon="document"

> “

Use Agent Memory from a Worker through an `agent_memory` binding.

<LinkTitleCard title="HTTP API" href="/agent-memory/api/http-api/" icon="document"

> “

Use Agent Memory from services that call the Cloudflare API directly.

<LinkTitleCard title="Limits" href="/agent-memory/platform/limits/" icon="document"

> “

Review validation limits and constraints for Agent Memory.

HTTP API

Use the HTTP API to call Agent Memory from services that do not run inside Cloudflare Workers. For Workers applications, use the Workers API through an `agent_memory` binding.

The HTTP API uses namespaces and profiles. A namespace scopes profiles for your application, and each profile is an isolated memory store. Profiles are created automatically when you first write to them.

Authentication

All requests require an API token with the appropriate Agent Memory permissions.

Include your API token in the `Authorization` header:

```
Authorization: Bearer <API_TOKEN>
```

For information about calling the Cloudflare API, refer to Make API calls.

Namespace management

A namespace is a top-level container that scopes memory profiles for your application.

Create a namespace

Creates a new namespace for the given account.

```
curl -X POST "https://api.cloudflare.com/client/v4/accounts/<ACCOUNT_ID>/agent-memory/namespaces" \
  -H "Authorization: Bearer <API_TOKEN>" \
  -H "Content-Type: application/json" \
  -d '{"name": "support-agent"}'
```

The response includes the namespace name that you use in Worker bindings and HTTP API calls.

```
{
  "result": {
    "id": "01JSGCEXAMPLE000000000000",
    "name": "support-agent",
    "created_at": "2026-04-21T00:00:00.000Z",
    "updated_at": "2026-04-21T00:00:00.000Z"
  },
  "success": true,
  "errors": [],
  "messages": []
}
```


List namespaces

Lists all namespaces for the given account. Results are paginated.

```
curl "https://api.cloudflare.com/client/v4/accounts/<ACCOUNT_ID>/agent-
memory/namespaces?per_page=50" \
  -H "Authorization: Bearer <API_TOKEN>"
```


```
{
  "result": [
    {
      "id": "01JSGCEXAMPLE000000000000",
      "name": "support-agent",
      "created_at": "2026-04-21T00:00:00.000Z",
      "updated_at": "2026-04-21T00:00:00.000Z"
    }
  ],
  "success": true,
  "errors": [],
  "messages": [],
  "result_info": {
    "cursor": "next-cursor",
    "per_page": 50,
    "count": 1
  }
}
```

Get a namespace

Retrieves a single namespace by name.

```
curl "https://api.cloudflare.com/client/v4/accounts/<ACCOUNT_ID>/agent-
memory/namespaces/<NAMESPACE_NAME>" \
  -H "Authorization: Bearer <API_TOKEN>"
```


```
{
  "result": {
    "id": "01JSGCEXAMPLE000000000000",
    "name": "support-agent",
    "created_at": "2026-04-21T00:00:00.000Z",
    "updated_at": "2026-04-21T00:00:00.000Z"
  },
  "success": true,
  "errors": [],
  "messages": []
}
```


Delete a namespace

Marks a namespace for deletion. The namespace name becomes available for reuse after deletion.

```
curl -X DELETE "https://api.cloudflare.com/client/v4/accounts/<ACCOUNT_ID>/agent-
memory/namespaces/<NAMESPACE_NAME>" \
  -H "Authorization: Bearer <API_TOKEN>"
```


```
{
  "result": null,
  "success": true,
  "errors": [],
  "messages": []
}
```


Profiles

Use profile endpoints to manage profiles and operate on memory stored in a named profile. Profiles are created automatically when you first write to them.

Delete a profile

Marks a profile and all its memories and messages for deletion.

```
curl -X DELETE "https://api.cloudflare.com/client/v4/accounts/<ACCOUNT_ID>/agent-memory/namespaces/<NAMESPACE_NAME>/profiles/<PROFILE_NAME>" \
  -H "Authorization: Bearer <API_TOKEN>"
```


```
{
  "result": null,
  "success": true,
  "errors": [],
  "messages": []
}
```


Delete a session

Marks all memories and messages in a profile that are tagged with the given session ID for deletion. Rows from other sessions in the same profile are untouched. Idempotent: deleting a session ID that has no rows is a no-op.

```
curl -X DELETE "https://api.cloudflare.com/client/v4/accounts/<ACCOUNT_ID>/agent-memory/namespaces/<NAMESPACE_NAME>/profiles/<PROFILE_NAME>/sessions/<SESSION_ID>" \
  -H "Authorization: Bearer <API_TOKEN>"
```


```
{
  "result": null,
  "success": true,
  "errors": [],
  "messages": []
}
```


Ingest messages

Processes a conversation and extracts structured memories from it. Agent Memory identifies facts, events, instructions, and tasks automatically, so you do not need to specify what to remember.

```
curl -X POST "https://api.cloudflare.com/client/v4/accounts/<ACCOUNT_ID>/agent-memory/namespaces/<NAMESPACE_NAME>/profiles/<PROFILE_NAME>/ingest" \
  -H "Authorization: Bearer <API_TOKEN>" \
  -H "Content-Type: application/json" \
  -d '{
    "messages": [
      {
        "role": "user",
        "content": "I prefer concise answers.",
        "timestamp": "2026-04-21T00:00:00.000Z"
      }
    ],
    "sessionId": "chat-2026-04-21"
  }'
```


```
{
  "result": null,
  "success": true,
  "errors": [],
  "messages": []
}
```


`ingest` is idempotent. Re-ingesting the same conversation does not create duplicate memories.

Remember a memory

Stores a single memory explicitly. Use `remember` when your application or agent already knows what should be stored.

```
curl -X POST "https://api.cloudflare.com/client/v4/accounts/<ACCOUNT_ID>/agent-memory/namespaces/<NAMESPACE_NAME>/profiles/<PROFILE_NAME>/remember" \
  -H "Authorization: Bearer <API_TOKEN>" \
  -H "Content-Type: application/json" \
  -d '{
    "content": "The user prefers concise answers.",
    "sessionId": "chat-2026-04-21"
  }'
```

```
{
  "result": {
    "id": "01JSGCEXAMPLE000000000000",
    "type": "instruction",
    "summary": "The user prefers concise answers.",
    "content": "The user prefers concise answers.",
    "sessionId": "chat-2026-04-21",
    "createdAt": "2026-04-21T00:00:00.000Z",
    "updatedAt": "2026-04-21T00:00:00.000Z"
  },
  "success": true,
  "errors": [],
  "messages": []
}
```


Recall memories

Searches stored memories in the profile and returns a synthesized answer grounded in the stored content.

```
curl -X POST "https://api.cloudflare.com/client/v4/accounts/<ACCOUNT_ID>/agent-memory/namespaces/<NAMESPACE_NAME>/profiles/<PROFILE_NAME>/recall" \
  -H "Authorization: Bearer <API_TOKEN>" \
  -H "Content-Type: application/json" \
  -d '{
    "query": "How should I answer this user?",
    "thinkingLevel": "low",
    "responseLength": "medium"
  }'
```


```
{
  "result": {
    "answer": "The user prefers concise answers.",
    "count": 1,
    "candidates": [
      {
        "id": "01JSGCEXAMPLE000000000000",
        "summary": "The user prefers concise answers.",
        "sessionId": "chat-2026-04-21",
```

```
        "score": 0.87
      }
    ]
  },
  "success": true,
  "errors": [],
  "messages": []
}
```


If no memories match the query, `recall` returns an empty answer.

List memories

Lists memories stored in the profile.

```
curl "https://api.cloudflare.com/client/v4/accounts/<ACCOUNT_ID>/agent-memory/namespaces/<NAMESPACE_NAME>/profiles/<PROFILE_NAME>/memories?per_page=50" \
  -H "Authorization: Bearer <API_TOKEN>"
```


```
{
  "result": [
    {
      "id": "01JSGCEXAMPLE000000000000",
      "type": "instruction",
      "summary": "The user prefers concise answers.",
      "sessionId": "chat-2026-04-21",
      "createdAt": "2026-04-21T00:00:00.000Z",
      "updatedAt": "2026-04-21T00:00:00.000Z"
    }
  ],
  "success": true,
  "errors": [],
  "messages": [],
  "result_info": {
    "cursor": "next-cursor",
    "per_page": 50,
    "count": 1
  }
}
```

List entries omit `content`. Use the get memory endpoint to retrieve the full memory.

To filter memories, use the `session_id` and `type` query parameters.

Get a memory

Retrieves a memory by ID.

```
curl "https://api.cloudflare.com/client/v4/accounts/<ACCOUNT_ID>/agent-
memory/namespaces/<NAMESPACE_NAME>/profiles/<PROFILE_NAME>/memories/<MEMORY_ID>" \
  -H "Authorization: Bearer <API_TOKEN>"
```


```
{
  "result": {
    "id": "01JSGCEXAMPLE000000000000",
    "type": "instruction",
    "summary": "The user prefers concise answers.",
    "content": "The user prefers concise answers.",
    "sessionId": "chat-2026-04-21",
    "createdAt": "2026-04-21T00:00:00.000Z",
    "updatedAt": "2026-04-21T00:00:00.000Z"
  },
  "success": true,
  "errors": [],
  "messages": []
}
```


Delete a memory

Deletes a memory by ID. Removes the memory and any source messages linked to it. Returns the deleted memory.

```
curl -X DELETE "https://api.cloudflare.com/client/v4/accounts/<ACCOUNT_ID>/agent-
memory/namespaces/<NAMESPACE_NAME>/profiles/<PROFILE_NAME>/memories/<MEMORY_ID>" \
  -H "Authorization: Bearer <API_TOKEN>"
```


```
{
  "result": {
    "id": "01JSGCEXAMPLE000000000000",
```

```
    "type": "instruction",
    "summary": "The user prefers concise answers.",
    "content": "The user prefers concise answers.",
    "sessionId": "chat-2026-04-21",
    "createdAt": "2026-04-21T00:00:00.000Z",
    "updatedAt": "2026-04-21T00:00:00.000Z"
  },
  "success": true,
  "errors": [],
  "messages": []
}
```


Get a summary

Generates a structured Markdown summary of everything stored in a memory profile. Use it to inspect what Agent Memory remembers about a profile.

```
curl -X POST "https://api.cloudflare.com/client/v4/accounts/<ACCOUNT_ID>/agent-memory/namespaces/<NAMESPACE_NAME>/profiles/<PROFILE_NAME>/summary" \
  -H "Authorization: Bearer <API_TOKEN>" \
  -H "Content-Type: application/json" \
  -d '{}'
```


```
{
  "result": {
    "summary": "## Summary\n\nThe user prefers concise answers."
  },
  "success": true,
  "errors": [],
  "messages": []
}
```


To scope the "Last Session" section of the summary, include the `sessionId` field in the request body.

Error responses

All endpoints return standard Cloudflare V4 error responses on failure:

```
{
  "result": null,
  "success": false,
  "errors": [
    {
      "code": 10008,
      "message": "Namespace name already exists"
    }
  ],
  "messages": []
}
```


Common error scenarios include:

Scenario	HTTP status
Invalid namespace name format	`400`
Authentication failure	`401`
Namespace name already exists	`409`
Namespace not found	`404`
Profile not found	`404`

API

import { DirectoryListing } from "~/components";

Use Agent Memory from a Worker through a binding, or from external services through the Cloudflare API.

Workers API

import { LinkCard, MetaInfo, Type, WranglerConfig } from "~/components";

Use the Workers API to access Agent Memory from your Worker. The binding connects your Worker to a namespace containing profiles, which are isolated memory stores for your agent.

Configure the binding

Add an `agent_memory` entry to your Wrangler configuration. The `binding` field is the variable name you use in Worker code, and the `namespace` field is the Agent Memory namespace to bind to.

```
[[agent_memory]]
binding = "MEMORY"
namespace = "<NAMESPACE_NAME>"
```

To bind multiple namespaces, add multiple entries to the `agent_memory` array.

Generated types

Run `npx wrangler types` to generate the binding type in `worker-configuration.d.ts`:

```
interface Env {
MEMORY: AgentMemoryNamespace;
}
```

Namespace methods

Use namespace methods on the binding to access and manage memory profiles.

`getProfile(profileName)`

Gets a memory profile by name. If the profile does not exist, Agent Memory creates it.

- `profileName` : Name of the profile to access. Maximum 100 characters.
- Returns

The first `getProfile()` call for a new profile may take longer while Agent Memory creates the profile.

`deleteProfile(profileName)`

Marks a profile and all its memories and messages for deletion.

- `profileName` : Name of the profile to delete. Maximum 100 characters.
- Returns

Profile methods

Call profile methods after you get a profile from the binding.

```
type AgentMemoryMemory = {
id: string;
type: "fact" | "event" | "instruction" | "task";
summary: string;
content: string;
sessionId: string | null;
createdAt: Date;
updatedAt: Date;
};
```

`ingest(messages, options?)`

Processes a conversation and extracts structured memories from it. Agent Memory identifies facts, events, instructions, and tasks automatically, so you do not need to specify what to remember.

- `messages` : Conversation messages to process.
- `options.sessionId` : Identifier for the conversation session. Maximum 64 characters. If omitted, Agent Memory derives one from the message content.
- Returns

```
type AgentMemoryMessage = {
role: "system" | "user" | "assistant";
content: string; // Max 32 KB
```

```
  timestamp?: Date;
};
```

`ingest()` is idempotent. Re-ingesting the same conversation does not create duplicate memories.

`remember(memory)`

Stores a single memory explicitly. Use `remember()` when your application or agent already knows what should be stored, instead of passing a conversation to `ingest()` for extraction.

- `memory.content` : Memory content to store. The service classifies and summarizes automatically.
- `memory.sessionId` : Identifier for the related conversation session.
- Returns

`recall(query, options?)`

Searches stored memories in the profile and returns a synthesized answer grounded in the stored content.

- `query` : Natural language question or search query. Maximum 1 KB (1,024 bytes UTF-8).
- `options.thinkingLevel` : Controls retrieval breadth. Higher levels search more candidates but take longer.
- `options.responseLength` : Controls the verbosity of the synthesized answer.
- `options.referenceDate` : Temporal anchor for date-relative queries.
- Returns

```
type AgentMemoryRecallResult = {
  count: number;
  answer: string;
  candidates: AgentMemoryScoredCandidate[];
};

type AgentMemoryScoredCandidate = {
  id: string;
  summary: string;
  sessionId: string | null;
  score: number;
};
```

If no memories match the query, `recall()` returns an empty answer.

`list(options?)`

Lists memories stored in the profile. Returns a paginated, filterable view of stored memories. Use the returned `cursor` (when present) to fetch the next page.

- `options.limit` : Maximum number of memories to return.
- `options.cursor` : Opaque cursor from a previous page.
- `options.sessionId` : Exact-match session filter.
- `options.type` : Exact-match memory-type filter.
- Returns

```
type AgentMemoryMemoryListEntry = Omit<AgentMemoryMemory, "content">;


type AgentMemoryListMemoriesResult = {
memories: AgentMemoryMemoryListEntry[];
cursor?: string;
};
```

List entries omit `content`. Use `get(memoryId)` to retrieve the full memory.

`get(memoryId)`

Retrieves a memory by ID.

- `memoryId` : Memory ID.
- Returns

Throws an error if the memory does not exist.

`delete(memoryId)`

Deletes a memory by ID. Removes the memory and any source messages linked to it. Returns the deleted memory.

- `memoryId` : Memory ID.
- Returns

Throws an error if the memory does not exist.

`deleteSession(sessionId)`

Marks all memories and messages in the profile that are tagged with the given session ID for deletion. Rows from other sessions in the same profile are untouched. Idempotent: deleting a session ID that has no rows is a no-op.

- `sessionId` : Session ID to delete. Maximum 64 characters.
- Returns

`getSummary(options?)`

Generates a structured Markdown summary of everything stored in a memory profile. Use it to inspect what Agent Memory remembers about a profile.

- `options.sessionId` : Session ID to scope the "Last Session" section of the summary. If omitted, Agent Memory uses the most recent session.
- Returns

```
type AgentMemoryGetSummaryResponse = {
summary: string;
};
```

Limits

Parameter	Limit
Messages per `ingest()` call	500
Message content size	32 KB (32,768 bytes UTF-8)
Session ID length	64 characters
`recall()` query size	1 KB (1,024 bytes UTF-8)

Refer to Limits for the complete list of constraints.

Next steps

How Agent Memory works

import { LinkCard } from "~/components";

Agent Memory is a managed service that gives your applications persistent, AI-powered memory. It automatically turns raw conversations into structured knowledge and retrieves the right context when you need it.

Memory types

Agent Memory classifies every extracted memory into one of four types:

- **Facts** — Stable knowledge about a person, project, or tool. Preferences, identities, relationships, and goals. Facts evolve over time through supersession: when a newer fact replaces an older one on the same topic, the old version is preserved but the latest surfaces in recall results.
- **Events** — Completed actions anchored to a point in time. Deployments, decisions, milestones, and observations. Events accumulate and do not conflict with each other.
- **Instructions** — Reusable procedures, workflows, and conventions. Like facts, instructions support supersession when updated.
- **Tasks** — Short-lived, session-scoped items such as active investigations and follow-ups. Tasks are deprioritized after the session ends.

How ingestion works

When you call `ingest()`, Agent Memory processes the conversation through several stages:

1. **Extraction** — AI reads the conversation and identifies discrete, memorable items. Each item is a standalone piece of knowledge with a clear summary and supporting content.
2. **Classification** — Each extracted item is classified into a memory type (fact, event, instruction, or task) and assigned a topic key, keywords, and search queries for later retrieval.
3. **Deduplication** — The system checks for duplicates against both the current batch and existing stored memories. Facts and instructions with the same topic key supersede older versions rather than creating duplicates.
4. **Storage** — Memories are written to durable storage with full-text search indexes. Non-task memories are also embedded as vectors for semantic search.

Raw conversation messages are always stored verbatim alongside extracted memories, preserving the original transcript for full-text search.

How recall works

When you call `recall()`, Agent Memory runs multiple retrieval strategies in parallel:

1. **Query analysis** — AI analyzes your query to determine the best retrieval approach, generating keyword terms, topic keys, and semantic search vectors.
2. **Parallel retrieval** — The system simultaneously searches across keyword indexes, topic key lookups, semantic vector indexes, and raw conversation messages.
3. **Scoring and ranking** — Results from all sources are combined and ranked to surface the most relevant memories while maintaining diversity across retrieval methods.
4. **Synthesis** — AI generates a natural language answer from the top-ranked memories, grounded in the actual stored content.

If no memories match the query, `recall()` returns an empty answer rather than hallucinating a response.

Idempotency and deduplication

Agent Memory is designed for safe re-ingestion:

- **Messages are content-addressed.** Each message gets a deterministic ID derived from its content and session. Sending the same message twice does not create a duplicate.
- **Sessions are deterministic.** If you do not provide a `sessionId`, one is derived from the message content. The same conversation always maps to the same session.
- **Facts and instructions evolve.** When a new memory shares a topic key with an existing one (for example, "editor preference"), the old memory is marked as superseded. The latest version surfaces in recall results, but the full history is preserved.

Related resources

Concepts

import { DirectoryListing } from "~/components";

Namespaces and profiles

Agent Memory uses a two-level isolation model: **namespaces** define memory domains, and **profiles** provide isolated memory stores for individual users, agents, teams, tenants, or application objects.

Namespaces

A namespace is a top-level container that scopes a set of memory profiles. Use namespaces to separate applications, environments, tenants, or memory layers such as user, team, and organization memory.

Profiles

A profile is an isolated memory store for a single entity. Each profile has its own stored memories and retrieval indexes.

Sessions

A session groups memories that come from the same interaction or conversation. Sessions are optional, but they make it easier to identify, inspect, and manage memories created from a specific conversation.

Sessions are scoped to a profile. Two different profiles can use the same session ID without conflict.

Isolation model

Conceptually, memories are scoped as `namespace > profile > memory`. No data crosses these boundaries:

```
Namespace: my-assistant-prod
  Profile: alice
    Memories
```

```
    Messages
  Profile: bob
    Memories
    Messages
```

A `recall()` on Alice's profile never returns memories from Bob's profile. Each profile is a self-contained memory system.

Platform

import { DirectoryListing } from "~/components";

Limits

The following limits apply to Agent Memory operations.

Feature	Limit
Messages per `ingest()` call	500
Message content size	32 KB (32,768 bytes UTF-8)
Recall query size	1 KB (1,024 bytes UTF-8)
Session ID length	64 characters
Profile name length	100 characters
Namespace name length	32 characters
List API page size	1 to 1,000 (default: 20)

Pricing

We are not currently billing for Agent Memory during private beta.

We will provide at least 30 days notice before we make any changes or start charging for Agent Memory usage.