Cloudflare Agents SDK

Version agents 0.32 · verified 2026-11-02

Stateful AI agents on Durable Objects — the Agent class, state and SQL, clients, scheduling, models and tools, MCP client and server, sandboxes, limits.

Contents

Mental model 2

The Agent class and routing 2

State and SQL 5

Clients and WebSockets 7

Scheduling and long-running work 9

Models and tools 11

MCP: client and server 13

Sandboxes and code execution 16

Limits and deployment 17

Further reading 18

Mental model

An agent built with the Agents SDK is a Durable Object with extra machinery installed: synced state, an embedded SQLite database, WebSocket connections to clients, a scheduler, and helpers for calling models and MCP servers. Every instance is one named, single-threaded actor that sleeps when idle and wakes on a request, message, or timer. Everything you know about Durable Objects — naming, hibernation, one object per unit of work — applies unchanged; the SDK only removes the plumbing.

The Agent class and routing

Piece	Purpose
`extends Agent` (from `agents`)	Defines the agent
`routeAgentRequest(req, env)`	Routes `/agents/:binding/:name`

Piece	Purpose
`getAgentByName(ns, name)`	Server-side RPC stub
`onStart(props)`	Runs on start or wake
`onRequest(req)`	Handles HTTP

Each class needs a Durable Object binding and an `exports` entry with SQLite storage. The URL's `:binding` is the binding name in kebab case — the binding `ChatAgent` is reached at `/agents/chat-agent/<name>`. From your own Worker code, `getAgentByName()` returns a stub; unlike plain `getByName()`, it starts the agent first, so `onStart()` has run.

```
import {
  Agent, routeAgentRequest, getAgentByName,
} from "agents";
export class Desk extends Agent {
  async ping() { return `pong ${this.name}`; }
}
```

```
export default {
  async fetch(req, env) {
    const r = await routeAgentRequest(req, env);
    if (r) return r;  // /agents/desk/*
    const a = await getAgentByName(
      env.Desk, "t1");
    return new Response(await a.ping());
  },
};
```

```
{
  "durable_objects": {
    "bindings": [
      { "name": "Desk", "class_name": "Desk" }
    ]
  },
  "exports": {
    "Desk": {
      "type": "durable-object",
      "storage": "sqlite"
```

```
    }
  }
}
```

> **Gotcha:** `routeAgentRequest()` routes any matching URL straight to the agent, so anyone who can guess `/agents/desk/<name>` can connect. Authenticate before routing, or in the agent's `onConnect()` and `onRequest()`.

State and SQL

Member	Does
`initialState`	Value before first write
`this.state`	Current value
`this.setState(s)`	Stores and broadcasts to clients
`validateStateChange(next, src)`	Throw to reject a change

Member	Does
`this.sql`...``	Tagged-template SQLite query

State is one JSON-serializable value per agent, stored in SQLite and pushed to every connected client; clients can push changes back, which pass through `validateStateChange()` first. Use `this.sql` for data that clients should not see or that grows: interpolated values become bound parameters, not string concatenation.

```
export class Notes extends Agent {
  initialState = { count: 0 };
  onStart() {
    this.sql`CREATE TABLE IF NOT EXISTS notes
      (id TEXT PRIMARY KEY, body TEXT)`;
  }
  add(id, body) {
    this.sql`INSERT INTO notes
      VALUES (${id}, ${body})`;
```

```
    const count = this.state.count + 1;
    this.setState({ count });
  }
}
```

> **Gotcha:** Calling `setState()` inside `onStateChanged()` re-triggers it. Return early when `source === "server"` or you build an infinite loop.

Clients and WebSockets

API	Side
`useAgent({ agent, name })`	React, from `agents/react`
`new AgentClient({...})`	Any JS, from `agents/client`
`onConnect` / `onMessage` / `onClose`	Server handlers
`this.broadcast(msg, without?)`	Send to all connections

API	Side
`@callable()` method	Client calls it via `agent.stub`

Clients hold a WebSocket to one agent instance. The agent hibernates between messages while sockets stay open, so keep anything you need after a wake in `this.state`, `connection.state`, or SQL, never in class fields. Methods marked with the `@callable()` decorator are invokable from the client; treat their arguments as untrusted input.

```
import { AgentClient } from "agents/client";

const client = new AgentClient({
  agent: "Notes",
  name: "user-42",
  host: "my-app.example.workers.dev",
  onStateUpdate: (s) => console.log(s.count),
```

```
});
await client.call("add", ["n1", "hello"]);
```

> **Note:** `@callable()` is a TC39 decorator. If the dev server fails with `SyntaxError: Invalid or unexpected token`, add the `agents/vite` plugin to supply the decorator transform.

Scheduling and long-running work

Call	Does
`this.schedule(30, "cb", data)`	Run in 30 seconds
`this.schedule(date, "cb")`	Run at a time
`this.schedule("0 8 * * *", "cb")`	Cron
`this.scheduleEvery(300, "cb")`	Every 300 seconds
`this.keepAliveWhile(fn)`	Blocks idle eviction
`this.runWorkflow("WF", params)`	Starts a Workflow

Schedules are rows in the agent's SQLite and survive restarts; the callback is the *name* of a method on the agent. An idle Durable Object is evicted after roughly 70 to 140 seconds, killing in-flight work, so wrap long model calls in `keepAliveWhile()`. That only prevents idle eviction — for work that must survive a deploy or crash, start a Workflow.

```
export class Reminder extends Agent {
  async onStart() {
    await this.schedule("0 8 * * *", "digest");
  }
  async remind(msg) {
    await this.schedule(3600, "send", { msg });
  }
  async send({ msg }) { console.log(msg); }
  async digest() {}
}
```

Models and tools

Piece	From
`createAI({ binding: env.AI })`	`agents/models/ai-sdk`
`ai("@cf/...")`	Workers AI model
`ai("anthropic/...")`	AI Gateway catalog model
`AIChatAgent`, `onChatMessage()`	`@cloudflare/ai-chat`
`tool({ inputSchema, execute })`	`ai` (AI SDK)

`createAI` routes Workers AI and third-party models through AI Gateway over the `AI` binding, so the Worker holds no vendor keys; it replaces the now-deprecated `workers-ai-provider` and `ai-gateway-provider`. `AIChatAgent` persists messages, resumes broken streams, and pairs with the `useAgentChat` React hook. Tools follow the AI SDK: `execute` runs on the server, and `stopWhen` bounds the loop.

```
import {
  AIChatAgent,
} from "@cloudflare/ai-chat";
import { createAI } from "agents/models/ai-sdk";
import { streamText, tool, stepCountIs,
  convertToModelMessages as toModel } from "ai";
import { z } from "zod";

const clock = tool({
  description: "Current UTC time",
  inputSchema: z.object({}),
  execute: async () => new Date().toISOString(),
});
```

```
export class Chat extends AIChatAgent {
  async onChatMessage() {
    const { AI } = this.env;
    const m = createAI({ binding: AI });
    const result = streamText({
      model: m("anthropic/claude-opus-5.5"),
```

```
      messages: await toModel(this.messages),
      tools: { clock },
      stopWhen: stepCountIs(5),  // max 5 steps
    });
    return result.toUIMessageStreamResponse();
  }
}
```

> **Gotcha:** AI Gateway catalog IDs use the catalog's spelling, `anthropic/claude-opus-5.5`, not the vendor's `claude-opus-5-5`. The vendor spelling only works when you pass the vendor's own model object.

MCP: client and server

API	Role
`this.addMcpServer(name, url)`	Connect as a client

API	Role
`result.state === "authenticating"`	Redirect to `result.authUrl`
`this.getMcpServers()`	Connections and tools
`createMcpHandler(factory)`	Stateless server, `agents/mcp/server`
`McpAgent`	Deprecated stateful server

As a client, an agent stores MCP connections in its SQLite and exposes their tools automatically; OAuth servers return an `authUrl` to send the user to. As a server, `createMcpHandler` serves an MCP SDK v2 server from a plain Worker, building a fresh server per request — no Durable Object needed. `McpAgent` is deprecated and feature-frozen; use it only for existing servers while they migrate.

```
import { McpServer } from
  "@modelcontextprotocol/server";
```

```
import { createMcpHandler } from
  "agents/mcp/server";
import { z } from "zod";
const input = z.object({ n: z.number() });
function mcp() {  // factory: one per request
  const s = new McpServer(
    { name: "demo", version: "1.0.0" });
  s.registerTool("sq", { inputSchema: input },
    async ({ n }) => ({ content: [
      { type: "text", text: `${n * n}` }] }));
  return s;
}
export default { fetch: createMcpHandler(mcp) };
```

> **Gotcha:** Pass the factory function, not a constructed `McpServer`. The handler builds a fresh server for every request; a single global instance is not supported.

Sandboxes and code execution

Option	Runs in
Container sandbox	Linux VM via `this.ctx.container`
Code Mode	A Dynamic Worker
`containers` in config	Attaches a container class

Cloudflare Containers and Sandboxes have been generally available since April 2026. Attaching a container to the agent's class gives each agent instance its own Linux sandbox, reached through `this.ctx.container`, for shell commands, compilers, and model-written code. The sandbox runs in its own VM: it cannot read the agent's storage, bindings, or environment, and reaches the internet only if allowed. When model-written code only calls methods you define, Code Mode in a Dynamic Worker starts faster.

```
{
  "containers": [
    {
      "class_name": "CodeAgent",
      "scheduling_policy": "durable_object"
    }
  ]
}
```

Limits and deployment

Limit	Value
CPU per agent	30 s, refreshed per request or message
Wall clock	Unlimited (waiting on an LLM is fine)
State per agent instance	1 GB
Concurrent agents	Tens of millions
Scripts per account	500 (Paid)

Agents inherit Workers and Durable Objects limits and pricing; deploy them with `wrangler deploy` like any Worker. The CPU budget resets on each HTTP request, WebSocket message, or scheduled task, so time spent awaiting a model does not count. The SDK is pre-1.0 and moves quickly — pin the version and read the changelog before upgrading.

```
npm i agents @cloudflare/ai-chat ai zod
npx wrangler types
npx wrangler deploy
```

Further reading

- Agents SDK documentation
- Agents API overview
- Chat agents
- MCP client API
- createMcpHandler
- Agents limits