Claude Agent SDK

Version @anthropic-ai/claude-agent-sdk 0.3.316 · verified 2026-11-02

Claude Code's agent loop as a TypeScript library — permissions, custom tools, subagents, hooks, sessions, streaming, budgets and hosting.

Contents

Mental model 2

Install and authenticate 2

The query loop 4

Tools and permissions 5

Custom tools and MCP servers 7

Subagents 8

Hooks 10

Sessions and streaming 12

Cost and token controls 14

Deployment 15

Further reading 16

Mental model

The Agent SDK is Claude Code as a library: `query()` starts the same agent loop — built-in tools, permissions, subagents, hooks, sessions, context compaction — and streams back every message it produces. You configure that loop; you do not write it. Under the hood the SDK spawns and supervises a `claude` subprocess with its own shell, working directory and session files, which is why hosting it looks like hosting a worker, not a stateless API call.

Install and authenticate

Need	Value
Package	`@anthropic-ai/claude-agent-sdk`
Runtime	Node.js 18.14+ (ESM)
Auth	`ANTHROPIC_API_KEY`
Amazon Bedrock	`CLAUDE_CODE_USE_BEDROCK=1`
Google Cloud	`CLAUDE_CODE_USE_VERTEX=1`

The package bundles a native Claude Code binary through npm optional dependencies, so no separate install is needed. It was renamed from the Claude Code SDK; the docs carry a migration guide. Products you ship authenticate with an API key or a cloud provider: offering claude.ai login needs Anthropic's prior approval.

```
npm pkg set type=module
npm install @anthropic-ai/claude-agent-sdk
export ANTHROPIC_API_KEY=sk-ant-...
npx tsx agent.ts
```

> **Gotcha:** The SDK does not read `.env` files, and `npm ci --omit=optional` skips the bundled binary. Both surface as "Not logged in" or a missing-executable error at the first `query()`.

The query loop

Message `type`	Carries
`system (init)`	Session ID, tools, MCP status
`assistant`	Text and `tool_use` blocks
`user`	Tool results fed back to Claude
`stream_event`	Token deltas (opt-in)
`result`	Final text, cost, usage, `session_id`

`query()` returns an async iterator. Each turn is one model response plus the tools it called; the loop ends when Claude answers with no tool call or a limit stops it. Check the result's `subtype` before reading `result`: only `success` has it.

```
import { query }
  from "@anthropic-ai/claude-agent-sdk";

for await (const msg of query({
  prompt: "Find and fix the crash in utils.ts",
  options: { allowedTools: ["Read", "Edit"] },
```

```
})) {
  if (msg.type !== "result") continue;
  console.log(msg.subtype === "success"
    ? msg.result
    : `stopped: ${msg.subtype}`);
}
```

> **Gotcha:** In single-shot mode the SDK throws after yielding an `error_*` result. Wrap the loop in `try` if the process must exit cleanly, and keep iterating past `result`: a few system events can follow it.

Tools and permissions

Option	Effect
`allowedTools`	Auto-approve listed calls
`disallowedTools`	Deny; a bare name hides the tool
`permissionMode`	`default`, `dontAsk`, `acceptEdits`, `plan`, `auto`, `bypassPermissions`

Option	Effect
`canUseTool`	Your callback for anything unresolved

Each tool call is checked in a fixed order: hooks, deny rules, ask rules, permission mode, allow rules, then `canUseTool`. A deny rule wins even in `bypassPermissions`. For an unattended agent, pair an allow-list with `dontAsk`, which denies anything that would prompt.

```
import type { Options }
  from "@anthropic-ai/claude-agent-sdk";

const locked: Options = {
  allowedTools: ["Read", "Glob", "Grep",
    "Bash(npm test *)"],
  disallowedTools: ["WebFetch"],
  permissionMode: "dontAsk",
};
```

> **Warning:** `allowedTools` does not limit `bypassPermissions`. `["Read"]` plus `bypassPermissions` still approves `Bash`, `Write` and `Edit`. Block tools with `disallowedTools` instead.

Custom tools and MCP servers

Piece	Purpose
`tool(name, desc, zodShape, fn)`	Define one tool
`createSdkMcpServer({ tools: [...] })`	In-process MCP server
`mcpServers: { key: server }`	Attach it, or stdio/HTTP servers
`mcp__<key>__<tool>`	The name to allow-list

Custom tools are MCP tools served in-process, so the same `mcpServers` option also takes external stdio and HTTP servers. Return `isError: true` with a useful message on failure; an uncaught throw reaches Claude as the raw exception text.

```
import { tool, createSdkMcpServer, query }
  from "@anthropic-ai/claude-agent-sdk";
import { z } from "zod";

const price = tool("get_price", "SKU price",
  { sku: z.string().describe("e.g. A-100") },
  async ({ sku }) => ({ content: [
    { type: "text", text: `${sku}: 9.99` }] }));
const shop = createSdkMcpServer({ name: "shop",
  version: "1.0.0", tools: [price] });
const run = query({ prompt: "Price of A-100?",
  options: { mcpServers: { shop },
    allowedTools: ["mcp__shop__get_price"] } });
```

Subagents

`AgentDefinition` field	Sets
`description`	When Claude should delegate
`prompt`	The subagent's system prompt
`tools`	Its own, narrower tool list

`AgentDefinition` field	Sets
`model`	e.g. `"sonnet"`, `"haiku"`

Claude delegates through the `Agent` tool, so never disallow it. A subagent runs in a fresh context: only the delegation prompt goes in, only its final message comes back. Use subagents to keep long searches out of the main context, run independent work in parallel, or give a reviewer read-only tools.

```
import type { Options }
  from "@anthropic-ai/claude-agent-sdk";

const opts: Options = {
  allowedTools: ["Read", "Grep", "Agent"],
  agents: {
    reviewer: {
      description: "Reviews diffs before a PR.",
      prompt: "Report correctness bugs only.",
      tools: ["Read", "Grep", "Glob"],
```

```
      model: "haiku",
    },
  },
};
```

> **Gotcha:** A subagent never sees the parent conversation. Paths, error text and decisions it needs must be in the delegation prompt, or it spends tokens rediscovering them.

Hooks

Event	Fires
`PreToolUse`	Before a tool runs; can deny or modify
`PostToolUse`	After a tool succeeds
`Stop`	When the agent tries to finish
`SubagentStop`	When a subagent finishes
`PreCompact`	Before history is summarised

Hooks are your code at fixed points in the loop, so they enforce what a prompt can only request. `PreToolUse` runs before every other permission step, and its deny holds even in `bypassPermissions`. The `matcher` is a pattern on the tool name.

```
import type { HookCallback }
  from "@anthropic-ai/claude-agent-sdk";

const noEnv: HookCallback = async (input) => {
  const ti = (input as { tool_input: object })
    .tool_input as { file_path?: string };
  const f = ti.file_path ?? "";
  if (!f.endsWith(".env")) return {};
  return { hookSpecificOutput: {
    hookEventName: "PreToolUse",
    permissionDecision: "deny",
    permissionDecisionReason: "No .env" } };
};
```

```
const hooks = { PreToolUse: [
  { matcher: "Write|Edit", hooks: [noEnv] }] };
```

Sessions and streaming

Option	Effect
`continue: true`	Resume the latest session in `cwd`
`resume: id`	Resume a specific session
`forkSession: true`	Branch a copy; original unchanged
`persistSession: false`	Keep nothing on disk
`includePartialMessages`	Emit `stream_event` deltas

Every `result` carries `session_id`, even on error, so you can resume after hitting a limit. Sessions are JSONL files under `~/.claude/projects/` on the machine that ran them.

```
import { query }
  from "@anthropic-ai/claude-agent-sdk";
```

```
let id = "";
const log = console.log;
const first = query({ prompt: "Audit auth/" });
for await (const m of first)
  if (m.type === "result") id = m.session_id;

const next = query({ prompt: "Fix the worst",
  options: { resume: id,
    includePartialMessages: true } });
for await (const m of next)
  if (m.type === "stream_event") log(m.event);
```

> **Gotcha:** Session files are local. An ephemeral container or a second host cannot `resume` unless you attach a `sessionStore` or copy the transcript file across.

Cost and token controls

Option	Stops or trims
`maxTurns`	Ends with `error_max_turns`
`maxBudgetUsd`	Ends with `error_max_budget_usd`
`effort`	`low` … `max` reasoning per turn
`model`	Cheaper model per agent or subagent

Both caps default to no limit, and the budget includes subagents' spend. `total_cost_usd` on the result is a client-side estimate from a bundled price table — use the Usage and Cost API for billing.

```
import type { Options }
  from "@anthropic-ai/claude-agent-sdk";


const capped: Options = {
  maxTurns: 20,
  maxBudgetUsd: 0.5,
  effort: "low",
};
```

> **Gotcha:** The budget is checked after each response arrives, so spend can pass `maxBudgetUsd` by one response plus whatever running subagents spend. Leave headroom.

Deployment

Concern	Guidance
Process model	One `claude` subprocess per agent
Starting size	1 GiB RAM, 5 GiB disk, 1 CPU per agent
Isolation	Sandboxed container per tenant
Config leakage	`settingSources: []`
Prompt	Default is minimal; preset is opt-in

By default `query()` reads the same filesystem config as the CLI: user, project and local settings, `CLAUDE.md`, skills, agents, and `.mcp.json` servers — which it loads without asking. On a shared host, run each tenant in its own filesystem and opt out. Use the `claude_code`

system-prompt preset only when you want Claude Code's behaviour.

```
import type { Options }
  from "@anthropic-ai/claude-agent-sdk";

const tenant: Options = {
  cwd: "/work/tenant-42",
  settingSources: [],
  env: { ...process.env,
    CLAUDE_CODE_DISABLE_AUTO_MEMORY: "1" },
  systemPrompt: { type: "preset",
    preset: "claude_code",
    append: "Answer tersely." },
};
```

Further reading

- Agent SDK overview
- How the agent loop works
- Configure permissions

- TypeScript SDK reference
- Hosting the Agent SDK
- Secure deployment