ChatGPT

OpenAI Agents SDK Documentation Overview

This report consolidates information from the **OpenAI Agents SDK for TypeScript** documentation into a single reference. It covers all guides and extensions from the site (https://openai.github.io/openai-agents-js/), summarizing concepts, code snippets and features. Citations link back to the original pages.

Overview

The OpenAI Agents SDK helps developers build **agentic AI applications** in TypeScript/JavaScript. It provides primitives such as **Agents**, **Tools** and **Handoffs**, enabling complex workflows with built-in tracing, guardrails, sessions and support for both text and real-time voice interactions [1] . Key features include:

- **Agent loop** – the SDK implements an iterative loop where the LLM is called, the response is inspected for final output, tool calls or handoffs, and the loop repeats until a stopping condition (final answer, error or max turns) [2] .
- **TypeScript-first** – strong typings via Zod/JSON schemas for tool inputs and agent outputs [3] .
- **Guardrails** – input/output/tool validations to enforce safety rules [4] .
- **Sessions** – persistent memory layer to preserve conversation history across runs [5] .
- **Tracing** – automatic collection of spans for debugging and performance analysis [6] .
- **Real-time voice** – integration with OpenAI's speech-to-speech models enabling streaming audio chat [7] .

Installation is via `npm install @openai/agents` (with `zod` dependency). A simple "Hello world" agent can be created by defining instructions and calling `run(agent, input)` [1] .

Quickstart

The quickstart guide walks through setting up a project:

1. **Initialize project** – use `npm create vite@latest` to scaffold a TypeScript project and install `@openai/agents` & `zod` 【2443362791021†L872-L1152】 .
2. **Set API key** – set `OPENAI_API_KEY` in your environment or call `setDefaultOpenAIKey()` [8] .
3. **Create an agent** – instantiate `Agent` with a `name` and `instructions`. Example:

```
const agent = new Agent({
  name: 'Writer',
  instructions: 'Write a haiku about cats.'
});
const answer = await run(agent, 'Compose a haiku');
```

1. **Add tools** – define function tools using `tool()` with Zod schemas for parameters; attach them to the agent. For example, a weather tool returns weather for a city [9] .
2. **Multi-agent orchestration** – create specialized agents (e.g., billing/refunds) and a triage agent that chooses the appropriate agent using handoffs. Use `Runner.run()` to orchestrate and inspect `result.output` and `result.history` 【2443362791021†L872-L1152】 .
3. **View traces** – the OpenAI dashboard can display traces recorded during the run for debugging and optimization 【2443362791021†L872-L1152】 .

Agents

An **Agent** represents an LLM configured with instructions, a model, tools and optional handoffs. Key aspects:

- **Configuration** – an agent is created with options: `name` , `instructions` , `prompt` / `promptId` , `handoffDescription` , `model` (default is `gpt-4.1`), `modelSettings` (temperature, topP, etc.), `tools` , `handoffs` and connectors such as MCP servers [3] .
- **Context** – two types of context exist: *local context* (dependency injection passed to tools via `RunContext`) and *agent/LLM context* (information visible to the LLM). Local context is not exposed to the LLM and can hold secrets or functions [10] .
- **Output types** – without a schema, agents return strings; with a Zod schema or JSON schema the output is structured; handoffs allow union output types [11] .
- **Multi-agent patterns** – agents can hand off sub-tasks to specialized agents (`transfer_to_<agent_name>` tool). Patterns include manager vs. specialist, sequential chains, parallel agents, critique–evaluate loops and code orchestrations [12] .

Running Agents

Agents are executed via `run()` or by constructing a `Runner` . Running an agent performs the loop: call the model, inspect response, process tool calls/handoffs, then continue until final output. Options include:

- **Streaming** – set `{ stream: true }` to get an async iterator of events. Use `toTextStream()` to extract only text. Interruptions (e.g., tool approval) can be resolved and the run resumed [13] .
- **Run arguments** – set `maxTurns` , `context` , `session` (for memory), `stream` , `tracing` , `errorHandlers` , `conversationId` and more [2] .
- **Run results** – `RunResult` holds `output` (final answer), `history` (inputs/outputs), `lastAgent` , `newItems` and `interruptions` (pending tool approvals) [14] . `StreamedRunResult` is an async iterator over events.

Results

The **Results** guide explains how to handle run outputs:

- Final output may be a string or typed object depending on agent configuration and schemas [11].
- Access history and new items via `result.history` and `result.newItems` to continue the conversation or resume after interruptions [14].
- `interruptions` array contains pending approvals for human-in-the-loop scenarios; use `state.approve()` or `state.reject()` to continue [14].

Tools

Tools allow agents to perform actions. Categories:

1. **Hosted OpenAI tools** – web search, file search, code interpreter and image generation (provided via helper functions) [15].
2. **Local built-in tools** – `computerTool`, `shellTool`, `applyPatchTool` enable interacting with the user's environment or applying patches to files [16].
3. **Function tools** – wrap arbitrary functions with JSON schema or Zod for structured input; pass options like `needsApproval` or `isEnabled` [9].
4. **Agents as tools** – expose another agent as a tool using `agent.asTool()`.
5. **MCP servers** – integrate remote tools via Model Context Protocol; connect hosted servers or local servers for retrieval and tool execution [17].
6. **Codex (experimental)** – call code generation models.

Each tool returns results via messages in the run history. Tools can be disabled or require human approval before execution.

Orchestrating Multiple Agents

The SDK supports two orchestration approaches:

- **Via the LLM** – supply a manager agent that decides which sub-agent to call using handoffs and tools. Provide clear instructions and recommended prompts; monitor responses and introspect reasoning [12].
- **Via code** – deterministically chain agents using structured outputs or loops; run agents in sequence or parallel; use a critique–evaluate loop where one agent critiques another's output; or process tasks in parallel to improve performance [12].

Handoffs

Handoffs allow an agent to delegate part of a conversation to another agent. To define handoffs:

- Specify `handoffs` array in the agent configuration or use `handoff()` helper to register a sub-agent [18].
- Each handoff creates a tool named `transfer_to_<agent_name>`; customizing the name and description is possible. Input types can be defined to control what data is passed during handoff [18].
- Use callbacks (`onHandoff`) for custom behaviors or to override default prompts and input filters.

Context Management

Context appears in two forms:

- **Local context** – an object passed to `run()` that holds dependencies or data for tools (e.g., user info, database connections). Tools access it via the `RunContext` argument. This context is *not* visible to the language model [10].
- **Agent/LLM context** – content visible to the LLM, provided via instructions, input to `run()`, retrieval tools or webs. Additional context can be added by retrieving data and injecting it into the LLM prompt [10].

Sessions

Sessions provide persistent memory across runs. The SDK offers:

- **MemorySession** – in-memory session for simple cases.
- **OpenAIConversationsSession** – connects to the Conversations API; supply options such as `conversationId`, `client`, `apiKey`, `baseURL` and `organization` [19].

A `Session` implementation stores conversation history; when passed to `Runner.run()` the runner automatically fetches previous items and persists new ones [5]. Sessions can be customized by implementing the `Session` interface.

Models

Agents call language models via the `Model` interface. Important points:

- **Default model** – if not specified, the SDK uses `gpt-4.1`. You can override via agent config, set `OPENAI_DEFAULT_MODEL` environment variable or define default model on `Runner` [20].
- **Model settings** – `ModelSettings` parameters (temperature, topP, frequency/presence penalties, tool choice, etc.) can be set per agent or run [21].

- **Prompt reuse** – pass `promptId` to use a server-stored prompt with variables; this allows centralizing prompt management [22].
- **Custom model providers** – implement `ModelProvider` and `Model` classes to wrap non-OpenAI LLMs; example `EchoProvider` echoes input and can be plugged into `Runner` 【627821135777929†L1218-L1303】.
- **Tracing exporter** – export traces by setting tracing API key using `setTracingExportApiKey()` [23].

Guardrails

Guardrails validate inputs, outputs or tool calls:

- **Input guardrails** – run before the model call; receive the user input and return `tripwireTriggered` and optional replacement text. If triggered, execution stops [24].
- **Output guardrails** – run on final agent responses to check for policy violations and optionally replace or block the output [25].
- **Tool guardrails** – wrap function tools to validate input/output; may allow, modify, reject or throw errors [26].

Guardrails can run in parallel or sequentially; sequential mode reduces token cost but increases latency [27]. Guardrail examples include blocking secrets, redacting PII, or verifying that tool output is appropriate 【599932972234342†L959-L1258】.

Streaming

Enabling streaming returns an async iterator of events for real-time updates:

- **Usage** – call `run(agent, input, { stream: true })` to receive a `StreamedRunResult` with `.toTextStream()` and `.completed` promise [13].
- **Event types** – events include `raw_model_stream_event` (model deltas), `run_item_stream_event` (run items like tool calls), and `agent_updated_stream_event` (agent updates) [28].
- **Human-in-the-loop** – during streaming, `stream.interruptions` lists pending tool approvals; call `state.approve()`/`state.reject()` and resume by re-running with `{ stream: true }` [29].
- **Tips** – always await `stream.completed`, re-specify `stream` when resuming, and use `toTextStream()` to get only text [30].

Human-in-the-Loop

To require user approval for tool calls:

- Set `needsApproval` to `true` or provide a function on the tool definition to dynamically decide if approval is required [31].
- When an agent attempts to call such a tool, execution pauses and `RunResult.interruptions` contains a `ToolApprovalItem`. The developer must approve or reject via `state.approve(item)` or `state.reject(item)` before resuming [32].
- In long approval scenarios, serialize `result.state` (e.g., `JSON.stringify(result.state)`) and later reconstruct with `RunState.fromString()` to resume [33].

Model Context Protocol (MCP)

The **Model Context Protocol** integrates external tool servers. Three server types are supported [17]:

1. **Hosted MCP servers** – remote servers used by OpenAI's Responses API. Connect using `hostedMcpTool()` and configure approval policies for tool calls. Use connectors to integrate with third-party services like Google Calendar by specifying `connectorId` and `authorization` [34].
2. **Streamable HTTP servers** – build a server exposing tools and retrieval; connect via `MCPServerStreamableHttp` with server URL [35].
3. **Stdio servers** – run tools via a local command (e.g., npx to run a server) using `MCPServerStdio`. You can define the command, encoding and environment [36].

Use `connectMcpServers()` to connect multiple servers; it returns lists of active and failed servers [37].

Tracing

Tracing collects spans for agent runs and is enabled by default on server environments [38]. Important points:

- **Export** – in most environments, traces are exported automatically; in browsers or Cloudflare Workers, call `getGlobalTraceProvider().forceFlush()` to manually flush [39].
- **Spans vs traces** – a trace represents one workflow; spans represent operations such as agent runs, model generations, tool calls, guardrails or handoffs [40].
- **Default tracing** – the SDK wraps `run()` and other operations into spans. Use `RunConfig.workflowName` or `withTrace()` to override the default name [6].
- **Custom traces** – use `withTrace()` or `getGlobalTraceProvider().createTrace()` to create nested traces and spans; add custom processors via `addTraceProcessor()` [41].

Configuration & Troubleshooting

- **API keys** – set `OPENAI_API_KEY` environment variable, call `setDefaultOpenAIKey()`, or customize the OpenAI client using `setDefaultOpenAIClient()` [42]. Switch between Chat Completions and Responses API with `setOpenAIAPI()` [42].
- **Tracing settings** – enable or disable tracing via `setTracingExportApiKey()` and `setTracingDisabled(true)` [43].
- **Debug logging** – enable verbose logs by setting `DEBUG=openai-agents*`; additional variables control whether model/tool data is logged [44].
- **Supported environments** – Node.js 22+, Deno, Bun are fully supported; Cloudflare Workers and browsers have limitations (manual flush for traces, restrictions on web sockets) [45]. Use debug logs to troubleshoot issues [46].
- **Versioning** – the package uses `0.Y.Z` semantics; Y increments indicate breaking changes, Z increments are new features/bug fixes [47].

Voice Agents

Voice Agents leverage OpenAI's speech-to-speech models for **real-time voice chat**. They support streaming audio, text and tool calls and are suitable for phone support, mobile apps or voice chat [7]. The Voice Agents SDK provides a TypeScript client for the **OpenAI Realtime API** [7].

Key Features [48]

- Connect over WebSocket or WebRTC.
- Works in both browser and backend environments.
- Supports audio and interruption handling.
- Enables multi-agent orchestration through handoffs.
- Defines and calls tools (function tools and hosted MCP tools).
- Supports custom guardrails to monitor model output.
- Provides callbacks for streamed events.
- Reuses components from text agents for voice agents.

Quickstart [49]

1. **Set up project** – create a Vite TypeScript app and install `@openai/agents` and `zod`. For browser-only usage, install `@openai/agents-realtime`.
2. **Generate client token** – since voice agents run in the browser, obtain an **ephemeral client key** from the Realtime API (starts with `ek_`). Use your OpenAI API key to request a client secret via `curl` [50].
3. **Create a** `RealtimeAgent` – similar to a regular agent; specify name and instructions [51].
4. **Create a** `RealtimeSession` – pass the agent and model (e.g., `gpt-realtime`). The session handles audio processing, interruptions and lifecycle [52].
5. **Connect to the session** – call `session.connect({ apiKey: 'ek_...' })` to connect via WebRTC (browser) or WebSocket (server) [53].

6. **Combine code** – import `RealtimeAgent` and `RealtimeSession`, create agent and session, and call `session.connect()` with the key. Grant microphone access; you can then speak to the agent [54].

Building Voice Agents

The `Building Voice Agents` guide dives deeper:

Session Configuration [55]

- Pass additional options when constructing `RealtimeSession` or calling `connect()`. Options include `model`, audio formats (`inputAudioFormat`, `outputAudioFormat`), transcription settings (`inputAudioTranscription.model`), local context, whether to store audio in history, output guardrails, tracing settings, group IDs, custom metadata, auto-trigger options and custom error formatters [55].

Handoffs [56]

- Voice agents can delegate tasks to other `RealtimeAgent`s similar to text agents. Handoffs update the ongoing session with the new agent configuration; input filters aren't applied and the model/ voice cannot be changed during a handoff [57].

Tools [58]

- Voice agents support **function tools** (executed locally) and **hosted MCP tools** (executed remotely). Define tools with `tool()` and attach them to a `RealtimeAgent`. Function tools run in the same environment (browser or server); sensitive actions should call a backend API [58]. Use `backgroundResult()` to return tool results without immediately triggering a model response [59]. Tool timeouts behave similarly to text agents [60].

Conversation History [61]

- Tools can access a snapshot of conversation history via the `details.context.history` parameter; note the last user phrase may not yet be transcribed [61].

Approval Before Execution [62]

- Setting `needsApproval: true` on a tool triggers a `tool_approval_requested` event. The developer should present a UI to approve or reject the call using `session.approve()` or `session.reject()` [62].

Guardrails [63]

- **RealtimeOutputGuardrail** functions run asynchronously during voice output to cut off responses that violate rules. They emit `guardrail_tripped` events. Guardrails run every 100 characters by default; adjust with `outputGuardrailSettings` [63].

Turn Detection & Interruptions [64]

- Voice activity detection determines when the user starts/stops speaking; configure via `turnDetection` with settings like `type`, `eagerness`, `createResponse` and `interruptResponse` [65].
- Interruptions occur when a user talks over the agent; the session emits `audio_interrupted`. Developers can also manually call `session.interrupt()` to stop audio playback [64].

Text Input & History Management [66]

- Send text via `session.sendMessage(text)` to provide additional context or enable multi-modal input [67].
- `session.history` stores the conversation; listen for `history_updated` events and use `session.updateHistory()` to modify or remove messages [68].

Limitations & Delegation [69]

- You cannot update function tool calls after execution; text output requires transcripts; truncated responses lack transcripts [70].
- Delegation through tools allows combining history and a tool call to offload complex tasks to another backend agent and return results to the user [71].

Transport Mechanisms

The **Realtime Transport Layer** guide explains how voice agents connect to the Realtime API over different transports [72]:

- **Default (WebRTC)** – audio is recorded via the microphone; to use your own stream or audio element, create an `OpenAIRealtimeWebRTC` instance and pass it as the transport [73].
- **WebSocket** – for server-side use cases (e.g., phone agents) pass `transport: 'websocket'` or instantiate `OpenAIRealtimeWebSocket` [74].
- **SIP** – use `OpenAIRealtimeSIP` to bridge calls from providers like Twilio. Steps include building initial config with `buildInitialConfig()`, attaching a `RealtimeSession` with the SIP transport, and connecting with the provider's `callId` [75].
- **Cloudflare note** – Cloudflare Workers cannot open WebSocket connections directly; use `CloudflareRealtimeTransportLayer` provided by the extensions package (described later) [76].
- **Building your own transport** – implement `RealtimeTransportLayer` and emit `RealtimeTransportEventTypes` to connect with alternative speech-to-speech APIs [77].

The guide also discusses interacting directly with the Realtime API by accessing the transport layer via `session.transport` or using it standalone (thin client) [78].

Extensions

Using Any Model with the AI SDK [79]

This extension integrates the **Vercel AI SDK** to use non-OpenAI models with the Agents SDK:

1. Install the extensions package: `npm install @openai/agents-extensions` [80].
2. Choose a model package from the AI SDK (e.g., `@ai-sdk/openai`) and install it [81].
3. Import the adapter and model:

```
import { openai } from '@ai-sdk/openai';
import { aisdk } from '@openai/agents-extensions/ai-sdk';
const model = aisdk(openai('gpt-5-mini'));
const agent = new Agent({ name: 'My Agent', instructions: '…', model });
run(agent, 'Question');
```

1. Provider metadata – pass provider-specific options via `providerData` (Agents SDK) which map to `providerMetadata` when using AI SDK [82].
2. UI helpers – `createAiSdkTextStreamResponse()` and `createAiSdkUiMessageStreamResponse()` wrap stream results into HTTP responses suitable for UI routes [83].

Connect Realtime Agents to Twilio [84]

- Twilio's **Media Streams API** sends raw audio from phone calls to a WebSocket server; this can drive a voice agent.
- Use the default Realtime transport in `websocket` mode, but it requires correct audio format and interruption timing.
- **TwilioRealtimeTransportLayer** (extensions package) simplifies the integration by handling connection, interruptions and audio forwarding [85].
- **Setup steps**:
- Ensure you have a Twilio account and phone number [86].
- Set up a WebSocket server (use ngrok or Cloudflare Tunnel to expose locally) [87].
- Install `@openai/agents-extensions` [88].
- Import `TwilioRealtimeTransportLayer` and create a `RealtimeAgent` & `RealtimeSession`, passing the transport [89].
- Call `session.connect({ apiKey: … })` to connect [90].
- **Tips** – create the transport layer as soon as you have the WebSocket connection; call `session.connect()` immediately; listen to `transport_event` events to inspect raw Twilio messages; enable debug logs with `DEBUG=openai-agents:extensions:twilio*` [91].
- A full example server using Fastify shows how to handle Twilio websockets and integrate tools like weather and secret functions [92].

Realtime Agents on Cloudflare [93]

- Cloudflare Workers cannot open outbound WebSockets using the global WebSocket constructor. The **CloudflareRealtimeTransportLayer** uses `fetch()` with `Upgrade: websocket` to work within workerd environments [93].
- **Setup**:
- Install `@openai/agents-extensions` [94].
- Import `CloudflareRealtimeTransportLayer`, `RealtimeAgent` and `RealtimeSession` [95].
- Create a transport specifying the realtime endpoint (e.g., `wss://api.openai.com/v1/realtime?model=gpt-realtime`) and attach it to the session [96].
- Connect the session with an API key [97].
- **Notes** – the Cloudflare transport uses `fetch()` with WebSocket upgrade; all `RealtimeSession` features (tools, guardrails) still work; enable debug logs for development [98].

Conclusion

The OpenAI Agents SDK for TypeScript offers a flexible framework for building both text- and voice-based agentic applications. It provides robust tooling (tools, guardrails, sessions, tracing), supports multi-agent orchestration and handoffs, and integrates with external services via MCP servers and extensions. The addition of **Voice Agents** and **Realtime transports** enables real-time speech-to-speech interactions over WebRTC, WebSocket, SIP, Twilio and Cloudflare environments, while the AI SDK adapter allows using non-OpenAI models. Guardrails, human-in-the-loop approvals and detailed tracing ensure safe, controlled operations. Overall, the SDK empowers developers to craft sophisticated agent workflows with strong observability and safety.

---

[1] OpenAI Agents SDK TypeScript | OpenAI Agents SDK
https://openai.github.io/openai-agents-js/

[2] Running agents | OpenAI Agents SDK
https://openai.github.io/openai-agents-js/guides/running-agents/

[3] Agents | OpenAI Agents SDK
https://openai.github.io/openai-agents-js/guides/agents/

[4] [24] [25] [26] [27] Guardrails | OpenAI Agents SDK
https://openai.github.io/openai-agents-js/guides/guardrails/

[5] [19] Sessions | OpenAI Agents SDK
https://openai.github.io/openai-agents-js/guides/sessions/

[6] [38] [39] [40] [41] Tracing | OpenAI Agents SDK
https://openai.github.io/openai-agents-js/guides/tracing/

[7] [48] Voice Agents | OpenAI Agents SDK
https://openai.github.io/openai-agents-js/guides/voice-agents/

[8] [20] [21] [22] [23] Models | OpenAI Agents SDK
https://openai.github.io/openai-agents-js/guides/models/

[9] [15] [16] Tools | OpenAI Agents SDK
https://openai.github.io/openai-agents-js/guides/tools/

[10] Context management | OpenAI Agents SDK
https://openai.github.io/openai-agents-js/guides/context/

[11] [14] Results | OpenAI Agents SDK
https://openai.github.io/openai-agents-js/guides/results/

[12] Orchestrating multiple agents | OpenAI Agents SDK
https://openai.github.io/openai-agents-js/guides/multi-agent/

[13] [28] [29] [30] Streaming | OpenAI Agents SDK
https://openai.github.io/openai-agents-js/guides/streaming/

[17] [34] [35] [36] [37] Model Context Protocol (MCP) | OpenAI Agents SDK
https://openai.github.io/openai-agents-js/guides/mcp/

[18] Handoffs | OpenAI Agents SDK
https://openai.github.io/openai-agents-js/guides/handoffs/

[31] [32] [33] Human in the loop | OpenAI Agents SDK
https://openai.github.io/openai-agents-js/guides/human-in-the-loop/

[42] [43] [44] Configuring the SDK | OpenAI Agents SDK
https://openai.github.io/openai-agents-js/guides/config/

[45] [46] Troubleshooting | OpenAI Agents SDK
https://openai.github.io/openai-agents-js/guides/troubleshooting/

[47] Release process | OpenAI Agents SDK
https://openai.github.io/openai-agents-js/guides/release/

[49] [50] [51] [52] [53] [54] Voice Agents Quickstart | OpenAI Agents SDK
https://openai.github.io/openai-agents-js/guides/voice-agents/quickstart/

[55] [56] [57] [58] [59] [60] [61] [62] [63] [64] [65] [66] [67] [68] [69] [70] [71] Building Voice Agents | OpenAI Agents SDK
https://openai.github.io/openai-agents-js/guides/voice-agents/build/

[72] [73] [74] [75] [76] [77] [78] Realtime Transport Layer | OpenAI Agents SDK
https://openai.github.io/openai-agents-js/guides/voice-agents/transport/

[79] [80] [81] [82] [83] Using any model with the Vercel's AI SDK | OpenAI Agents SDK
https://openai.github.io/openai-agents-js/extensions/ai-sdk/

[84] [85] [86] [87] [88] [89] [90] [91] [92] Connect Realtime Agents to Twilio | OpenAI Agents SDK
https://openai.github.io/openai-agents-js/extensions/twilio/

[93] [94] [95] [96] [97] [98] Realtime Agents on Cloudflare | OpenAI Agents SDK
https://openai.github.io/openai-agents-js/extensions/cloudflare/