## 1. Creating an Agent

An **Agent** in the OpenAI Agents SDK is essentially an AI model (LLM) configured with a specific role or behavior. You define an agent by giving it a **name** and a set of **instructions** (a system prompt describing its role)[[1]](https://openai.github.io/openai-agents-js/guides/quickstart/#:~:text=Agents%20are%20defined%20with%20instructions,and%20a%20name). In simple terms, the instructions tell the agent “who it is” and how to respond. Once an agent is created, you can pass user queries to it and get responses. Below is a minimal example of creating a travel assistant agent and running it to get an answer.

**Example (agentCreation.ts):**

import { Agent, run } from '@openai/agents';  
  
const travelAgent = new Agent({  
 name: 'Travel Guide',  
 instructions: 'You are a helpful travel assistant.', // system prompt defining the agent  
});  
  
// Use the agent to answer a question:  
const result = await run(travelAgent, 'What is a good place to visit in July?');  
console.log(result.finalOutput);

**Instruction could accept function as well as string .**

## 2. Custom vs. Hosted Tools

Agents can use **tools** to perform actions or look up information beyond just chatting. Think of tools as extra skills or apps the agent can invoke (for example, searching the web or calling a calculator). There are two main categories of tools:

* **Hosted tools:** Built-in tools that run on OpenAI’s servers (e.g. web search, file/database search, code execution, image generation)[[2]](https://openai.github.io/openai-agents-js/guides/tools/#:~:text=1,server%20running%20on%20your%20machine). You enable these by adding them to the agent, and the heavy work is handled remotely (for example, a 'web\_search' tool can let the agent do internet searches).
* **Function (custom) tools:** Your own functions that you wrap for the agent to use[[3]](https://openai.github.io/openai-agents-js/guides/tools/#:~:text=You%20can%20turn%20any%20function,helper). You define what they do and their input schema. The agent can call these functions when it needs that capability. For instance, you might create a custom tool to look up weather or local travel recommendations.

Using the SDK, you simply attach tools when creating the agent. In the example below, our travel agent has one hosted tool (web search) and one custom tool for providing a fun fact about a destination. The agent can decide to use these tools while answering the user’s question.

**Example (tools.ts):**

import { Agent, run, tool, webSearchTool } from '@openai/agents';  
import { z } from 'zod';  
  
// A custom function tool definition (using zod for parameter schema):  
const destinationFactTool = tool({  
 name: 'destination\_fact',  
 description: 'Provide a fun fact about a given travel destination',  
 parameters: z.object({ destination: z.string() }), // expects a destination name  
 async execute({ destination }) {  
 // For simplicity, return a canned fun fact. In real use, you might query a database or API.  
 return `${destination} is known for its amazing culture and history.`;  
 },  
});  
  
const travelAgent = new Agent({  
 name: 'Travel Assistant',  
 instructions: 'You help users with travel plans and information.',  
 tools: [  
 webSearchTool(), // a hosted tool for internet search  
 destinationFactTool // our custom tool defined above  
 ],  
});  
  
// Now the agent can use these tools when needed.  
const result = await run(travelAgent, 'Tell me a fun fact about Paris.');  
console.log(result.finalOutput);

## 3. Multi-Agent Systems and Handoffs

The SDK supports **multi-agent systems**, where one agent can delegate tasks to other specialized agents. A *handoff* is when an agent passes the conversation or a sub-task to another agent better suited for it[[4]](https://openai.github.io/openai-agents-js/guides/handoffs/#:~:text=Handoffs%20let%20an%20agent%20delegate,handle%20bookings%2C%20refunds%20or%20FAQs). This setup is useful if you want to break down a complex problem or have domain-specific experts. For example, in a travel app, you might have one agent specialize in flights and another in hotels, and a top-level agent that decides which one should handle the user’s request.

To enable this, you can provide a list of sub-agents in an agent’s configuration via the handoffs property. The gateway agent will automatically transfer control to one of these sub-agents when appropriate[[5]](https://openai.github.io/openai-agents-js/guides/handoffs/#:~:text=Every%20agent%20accepts%20a%20,helper) (the decision is made by the model’s reasoning based on the instructions). Below is an example with a travel planner agent that can hand off to a Flights Agent or a Hotels Agent depending on the query.

**Example (multiAgent.ts):**

import { Agent, run } from '@openai/agents';  
  
// Two specialized agents for flights and hotels:  
const flightsAgent = new Agent({  
 name: 'Flights Agent',  
 instructions: 'You assist with flight inquiries and bookings.',  
});  
const hotelsAgent = new Agent({  
 name: 'Hotels Agent',  
 instructions: 'You assist with hotel inquiries and reservations.',  
});  
  
// Main agent that can delegate to the above agents:  
const travelAgent = new Agent({  
 name: 'Travel Planner',  
 instructions: 'Decide which agent (flights or hotels) should handle the user request and hand it off.',  
 handoffs: [flightsAgent, hotelsAgent] // sub-agents to delegate to  
});  
  
// Run the main agent with a query. It will hand off to the appropriate sub-agent:  
const result = await run(travelAgent, 'I need to book a hotel in Paris.');  
console.log(`Final answer from ${result.finalAgent?.name}: ${result.finalOutput}`);

*(In this example, the Travel Planner agent will recognize the user is asking about a hotel, and internally transfer the question to the Hotels Agent. The result.finalAgent indicates which agent produced the final answer.)*

## 4. Context Management

**Context management** is about providing background information or state that the agent can use during a conversation. In the Agents SDK, “context” can mean two things:

1. **Local context:** Data your code supplies that tools and hooks can access, but the AI model itself does *not* directly see[[6]](https://openai.github.io/openai-agents-js/guides/context/#:~:text=Context%20is%20an%20overloaded%20term,context%20you%20might%20care%20about). This could be user-specific info (like name, preferences, IDs) or external dependencies. You pass this in when you run the agent, and it’s carried through the process for use by your custom tools or callbacks.

Eg : UserID , chatID

1. **Agent/LLM context:** Information included in the conversation that *is* visible to the model[[6]](https://openai.github.io/openai-agents-js/guides/context/#:~:text=Context%20is%20an%20overloaded%20term,context%20you%20might%20care%20about). This includes the agent’s instructions (system prompt), the conversation history, and any information the agent retrieves via tools. You manage this by what you put in the instructions or by feeding relevant data through the conversation (or using retrieval tools).

**(instructions + messages + retrieved content**

**Memories :** Information shared by the user

 **Facts** → explicit identifiers (name, city, email, home airport, passport country, loyalty IDs). *Always upsert KV.*

 **Semantic** → likes/dislikes/prefs (diet, style, budget level, seat, hotel type, interests). *Store atomic notes in vector + mirror critical ones in KV.*

 **Episodic** → dates, companions, trip intent, booking details. *Vector with chat\_id & TTL.*

User Query : My name is Sam and I live in New York. I love hiking and photography.

**Fact**

* name: "Sam"
* home\_city: "New York"

**Semantic**

* interest: "hiking"
* interest: "photography"
* User query : Plan a 5-day Italy trip in October, mid budget. I love food & history, vegetarian, prefer boutique  
    
  **Semantic**
  + diet: "vegetarian"
  + interests: ["food","history"]
* **Episodic**
  + trip\_request: { destination: "Italy", month: "October", duration\_days: 5,

In practice, to use local context you define a context object and provide it when calling run(). The SDK wraps it in a RunContext so that any tool’s execute function (or handoff hook) can access runContext.context during that run[[7]](https://openai.github.io/openai-agents-js/guides/context/#:~:text=Local%20context%20is%20represented%20by,from%20or%20modify%20that%20object). The model itself won’t see this object unless a tool explicitly uses it to produce output. This is useful for things like personalization. For example, you might store the user’s home airport or loyalty status in context so that a travel agent can tailor answers to the user.

Below, we show an agent with a simple user context. We pass in a user’s name and home airport as context, and we define a tool that can use this context to answer a question about the user.

**Example (contextManagement.ts):**

import { Agent, run, tool, RunContext } from '@openai/agents';  
import { z } from 'zod';  
  
// Define an interface for the user info we want to keep in context:  
interface UserInfo {  
 name: string;  
 homeAirport: string;  
}  
  
// A custom tool that returns the current user's home airport from the context  
const getHomeAirport = tool({  
 name: 'get\_home\_airport',  
 description: 'Returns the home airport of the current user',  
 parameters: z.object({}), // no input needed for this tool  
 async execute(\_args: {}, runContext?: RunContext<UserInfo>) {  
 // Access the context to retrieve user info:  
 const airport = runContext?.context.homeAirport;  
 return `The user's home airport is ${airport}.`;  
 },  
});  
  
const travelAgent = new Agent<UserInfo>({  
 name: 'Travel Assistant',  
 instructions: 'You help with personalized travel information.',  
 tools: [getHomeAirport]  
});  
  
// Prepare some context data for the user:  
const userInfo: UserInfo = { name: 'Alice', homeAirport: 'JFK International Airport' };  
  
// Run the agent with the user context. The agent (via the tool) can use this data in its answer.  
const result = await run(travelAgent, 'What is my home airport?', { context: userInfo });  
console.log(result.finalOutput); // e.g. "The user's home airport is JFK International Airport."

*(Here, the agent uses the getHomeAirport tool which reads from the provided context instead of relying on the language model’s knowledge.)*

## 5. Guardrails

**Guardrails** are safety checks or rules that you can enforce on the agent’s inputs and outputs. They run in parallel with the agent’s normal reasoning to catch unwanted content or mistakes. For example, you might use guardrails to prevent the agent from answering certain types of questions or to validate its answers. If a guardrail condition is triggered, it will stop the agent from proceeding. In fact, the SDK will throw an error and halt execution if a guardrail detects a problem[[8]](https://openai.github.io/openai-agents-js/guides/guardrails/#:~:text=Guardrails%20run%20in%20parallel%20to,the%20costly%20model%20from%20running).

There are two kinds of guardrails[[9]](https://openai.github.io/openai-agents-js/guides/guardrails/#:~:text=There%20are%20two%20kinds%20of,guardrails):

* **Input guardrails:** These run **before** the agent responds, examining the user’s input. If the input is disallowed or problematic, the agent won’t even attempt to answer.
* **Output guardrails:** These run **after** the agent produces an answer (but before it’s returned to the user). They can inspect the final output and decide if it should be blocked or altered.

Under the hood, a guardrail is simply a function that returns a result indicating whether a certain condition (a “**tripwire**”) has been met. If the tripwire is triggered, the SDK throws a specific exception (and no answer is returned)[[10]](https://openai.github.io/openai-agents-js/guides/guardrails/#:~:text=Section%20titled%20%E2%80%9CTripwires%E2%80%9D). You can then catch that exception in your code and handle it (for example, by showing an error or a safer response to the user).

To implement guardrails, you attach them to the agent when creating it using the inputGuardrails or outputGuardrails options. Below is a simple example of an **input guardrail** in a travel context. This guardrail checks the user’s question and if it contains certain forbidden terms (like asking about traveling to a war zone or something illegal), it triggers and stops the agent from responding.

**Example (guardrails.ts):**

import { Agent, run, InputGuardrail, InputGuardrailTripwireTriggered } from '@openai/agents';  
  
// Define an input guardrail that blocks queries containing certain forbidden keywords:  
const safetyGuardrail: InputGuardrail = {  
 name: 'SafetyGuardrail',  
 async execute({ input }) {  
 const forbiddenTopics = ['war zone', 'illegal']; // topics we want to disallow  
 const lowered = input.toLowerCase();  
 const foundForbidden = forbiddenTopics.some(topic => lowered.includes(topic));  
 return {  
 outputInfo: null, // no additional info to return  
 tripwireTriggered: foundForbidden // true if input contains forbidden topic  
 };  
 }  
};  
  
const travelAgent = new Agent({  
 name: 'Cautious Travel Planner',  
 instructions: 'You help plan trips while avoiding unsafe or disallowed requests.',  
 inputGuardrails: [safetyGuardrail]  
});  
  
try {  
 // This query contains "war zone", which our guardrail should catch and trigger on:  
 await run(travelAgent, 'Can you plan a vacation in a war zone?');  
 console.log("Guardrail didn't trip (unexpected)");  
} catch (e) {  
 if (e instanceof InputGuardrailTripwireTriggered) {  
 console.log('Guardrail triggered: unsafe request was blocked.');  
 }  
}

*(In this example, the guardrail will detect the phrase "war zone" and trigger a tripwire, causing the run to throw an InputGuardrailTripwireTriggered error. We catch the error to confirm that the guardrail worked.)*

## 6. Streaming

By default, when you run an agent, you get the final answer only after all the reasoning and tool usage are done. **Streaming** mode lets you receive the agent’s response gradually, as it’s being generated. This can make the app feel more responsive, because the user starts seeing the answer unfold in real time instead of waiting for the whole answer. The Agents SDK supports streaming by delivering output incrementally[[11]](https://openai.github.io/openai-agents-js/guides/streaming/#:~:text=The%20Agents%20SDK%20can%20deliver,result%20before%20updating%20the%20user).

To use streaming, you simply pass a stream: true option when calling run(). Instead of returning a complete result immediately, the run function will return a streaming object that you can iterate over to get pieces of the output as they arrive[[12]](https://openai.github.io/openai-agents-js/guides/streaming/#:~:text=Section%20titled%20%E2%80%9CEnabling%20streaming%E2%80%9D). You can think of this like an async iterator or a readable stream of events. Most importantly, you can extract the text chunks and update your UI or logs continuously.

In the example below, we create a travel storytelling agent. We enable streaming and then loop through the text stream to print the story as it’s generated, chunk by chunk.

**Example (streaming.ts):**

import { Agent, run } from '@openai/agents';  
  
const travelStoryAgent = new Agent({  
 name: 'Travel Storyteller',  
 instructions: 'You share engaging travel stories about given destinations.',  
});  
  
const stream = await run(travelStoryAgent, 'Tell me a story about visiting Rome.', { stream: true });  
  
// Output the response text in real-time as the agent generates it:  
for await (const textChunk of stream.toTextStream()) {  
 process.stdout.write(textChunk);  
}  
  
// Wait for the stream to fully finish (ensures all chunks are processed)  
await stream.completed;  
console.log('\n[Streaming finished]');

**Sources:** The above explanations and examples are based on the OpenAI Agents SDK documentation, including the guides on defining agents[[1]](https://openai.github.io/openai-agents-js/guides/quickstart/#:~:text=Agents%20are%20defined%20with%20instructions,and%20a%20name), using tools[[2]](https://openai.github.io/openai-agents-js/guides/tools/#:~:text=1,server%20running%20on%20your%20machine)[[3]](https://openai.github.io/openai-agents-js/guides/tools/#:~:text=You%20can%20turn%20any%20function,helper), multi-agent orchestration and handoffs[[4]](https://openai.github.io/openai-agents-js/guides/handoffs/#:~:text=Handoffs%20let%20an%20agent%20delegate,handle%20bookings%2C%20refunds%20or%20FAQs)[[5]](https://openai.github.io/openai-agents-js/guides/handoffs/#:~:text=Every%20agent%20accepts%20a%20,helper), context management[[6]](https://openai.github.io/openai-agents-js/guides/context/#:~:text=Context%20is%20an%20overloaded%20term,context%20you%20might%20care%20about)[[7]](https://openai.github.io/openai-agents-js/guides/context/#:~:text=Local%20context%20is%20represented%20by,from%20or%20modify%20that%20object), guardrails for input/output safety[[9]](https://openai.github.io/openai-agents-js/guides/guardrails/#:~:text=There%20are%20two%20kinds%20of,guardrails)[[10]](https://openai.github.io/openai-agents-js/guides/guardrails/#:~:text=Section%20titled%20%E2%80%9CTripwires%E2%80%9D), and streaming responses[[11]](https://openai.github.io/openai-agents-js/guides/streaming/#:~:text=The%20Agents%20SDK%20can%20deliver,result%20before%20updating%20the%20user)[[12]](https://openai.github.io/openai-agents-js/guides/streaming/#:~:text=Section%20titled%20%E2%80%9CEnabling%20streaming%E2%80%9D). These resources provide more detail and additional examples for each topic.

7. Memories

“Memories” are small, useful facts you save about a user outside the model—like sticky notes in your own database (e.g., mem0 + a vector store). You add durable details (name, home airport, budget style, prefers non-stop/veg, past trips), then on each new query you retrieve the top relevant notes and either summarize them into the prompt (so the model can reason with them) or use them privately in your tools (so the model never sees raw data). They’re not the model’s built-in recall; they only work because you fetch and inject them at answer time. The payoff is personalization and continuity, and the rule of thumb is: store minimally, redact sensitive info, and inject only short, safe summaries.

Code Example :   
import 'dotenv/config';

import { Agent, RunContext, run } from '@openai/agents';

import { Memory } from 'mem0ai/oss';

const mem = new Memory({

  version: 'v1.1',

  enableGraph: false,

  vectorStore: {

    provider: 'redis',

    config: {

      url: 'redis://localhost:6379/0',   // pin DB 0

      indexName: 'memories\_idx\_2',         // <- match your Redis index

      embeddingModelDims: 1536,

    },

  },

});

interface UserContext {

  userId: string;

  query: string;

}

async function buildInstructions(runContext: RunContext<UserContext>) {

  console.log('Searching memories for:', runContext.context.userId, 'q=', runContext.context.query);

  let memories;

  try {

    memories = await mem.search(runContext.context.query, {

      userId: runContext.context.userId,

    });

    console.log('mem.search ok:', memories);

  } catch (e) {

    console.error('mem.search error:', e);

    memories = { results: [] };

  }

  const memStr = (memories.results ?? []).map((m: any) => m.memory).join('\n');

  return `

    You are a friendly assistant.

    Context about user (from memory):

    ${memStr || 'No relevant past info'}

  `;

}

const agent = new Agent<UserContext>({

  name: 'MemoryAwareAgent',

  instructions: buildInstructions,

});

async function runChat(userId: string, query: string) {

  // @ts-ignore: run helper executes the agent

  const result = await run(agent, query, { context: { userId, query } });

  const reply: string = result.finalOutput || 'agent response missing';

  console.log(`User: ${query}`);

  console.log(`Agent: ${reply}`);

  try {

    await mem.add(

      [

        { role: 'user', content: query },

        { role: 'assistant', content: reply },

      ],

      { userId }

    );

    console.log('mem.add ok');

  } catch (e) {

    console.error('mem.add error:', e);

  }

}

function generateRandom10DigitNumber() {

  // Define the minimum and maximum values for a 10-digit number.

  const min = 1000000000; // Smallest 10-digit number

  const max = 9999999999; // Largest 10-digit number

  return `${Math.floor(Math.random() \* (max - min + 1)) + min}`;

}

(async () => {

    const idx  = generateRandom10DigitNumber();

  await runChat( idx, 'My name is Sam and I live in New York. I love hiking and photography.');

  await new Promise((r) => setTimeout(r, 4000));

  await runChat(idx, 'what is the old memory u have about me?');

})();

[[1]](https://openai.github.io/openai-agents-js/guides/quickstart/" \l ":~:text=Agents%20are%20defined%20with%20instructions,and%20a%20name) Quickstart | OpenAI Agents SDK

<https://openai.github.io/openai-agents-js/guides/quickstart/>

[[2]](https://openai.github.io/openai-agents-js/guides/tools/#:~:text=1,server%20running%20on%20your%20machine) [[3]](https://openai.github.io/openai-agents-js/guides/tools/#:~:text=You%20can%20turn%20any%20function,helper) Tools | OpenAI Agents SDK

<https://openai.github.io/openai-agents-js/guides/tools/>

[[4]](https://openai.github.io/openai-agents-js/guides/handoffs/#:~:text=Handoffs%20let%20an%20agent%20delegate,handle%20bookings%2C%20refunds%20or%20FAQs) [[5]](https://openai.github.io/openai-agents-js/guides/handoffs/#:~:text=Every%20agent%20accepts%20a%20,helper) Handoffs | OpenAI Agents SDK

<https://openai.github.io/openai-agents-js/guides/handoffs/>

[[6]](https://openai.github.io/openai-agents-js/guides/context/#:~:text=Context%20is%20an%20overloaded%20term,context%20you%20might%20care%20about) [[7]](https://openai.github.io/openai-agents-js/guides/context/#:~:text=Local%20context%20is%20represented%20by,from%20or%20modify%20that%20object) Context management | OpenAI Agents SDK

<https://openai.github.io/openai-agents-js/guides/context/>

[[8]](https://openai.github.io/openai-agents-js/guides/guardrails/#:~:text=Guardrails%20run%20in%20parallel%20to,the%20costly%20model%20from%20running) [[9]](https://openai.github.io/openai-agents-js/guides/guardrails/#:~:text=There%20are%20two%20kinds%20of,guardrails) [[10]](https://openai.github.io/openai-agents-js/guides/guardrails/#:~:text=Section%20titled%20%E2%80%9CTripwires%E2%80%9D) Guardrails | OpenAI Agents SDK

<https://openai.github.io/openai-agents-js/guides/guardrails/>

[[11]](https://openai.github.io/openai-agents-js/guides/streaming/#:~:text=The%20Agents%20SDK%20can%20deliver,result%20before%20updating%20the%20user) [[12]](https://openai.github.io/openai-agents-js/guides/streaming/#:~:text=Section%20titled%20%E2%80%9CEnabling%20streaming%E2%80%9D) Streaming | OpenAI Agents SDK

<https://openai.github.io/openai-agents-js/guides/streaming/>