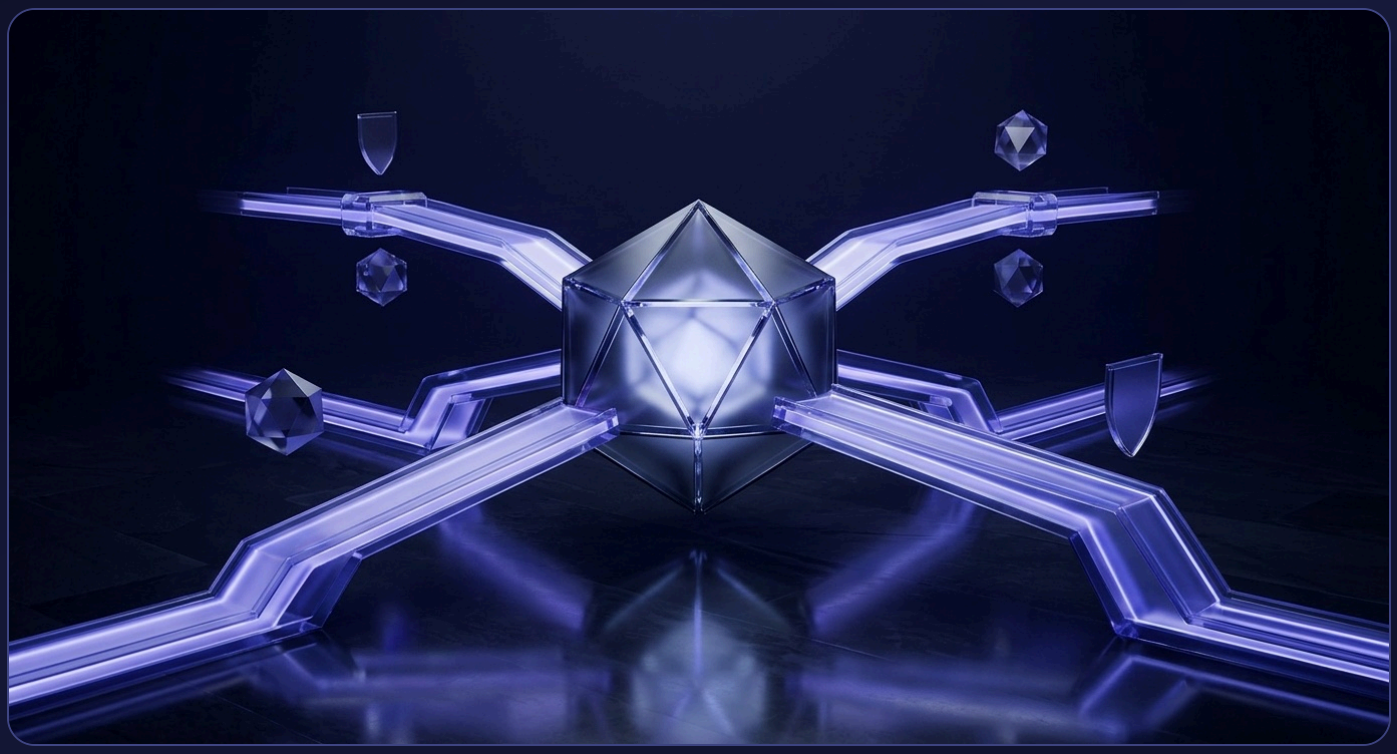
OPENAI CODEX, AGENTS & REALTIME STUDIOS • FIELD GUIDE

OpenAI Agents API & SDK

From minimal Python and TypeScript primitives to production multi-agent systems—mastering specialist handoffs, typed tool contracts, parallel guardrails, Redis memory, MCP connectors, and live trace observability.

Python & TypeScript | Managed Agents API | Handoffs & Tools | Parallel Guardrails | Built-In Tracing | Redis & MCP State



Minimal Core Primitives

Build deterministic agent loops using just `Agent`, `Runner`, tools, handoffs, and guardrails—with zero bloated abstractions.

Parallel Safety & Traces

Run fast input/output guardrails alongside model turns and inspect every span, latency spike, and token cost out of the box.

Portable State & MCP

Graduate seamlessly from local `SQLiteSession` prototypes to Redis clusters and persistent MCP servers or the hosted Agents API.

Minimal Primitives, Production Harness

OpenAI's **Agents SDK**—published as `openai-agents` for Python and `@openai/agents` for TypeScript—turns the experimental simplicity of Swarm into a battle-tested runtime, paired with the managed **Agents API** when you want OpenAI to host the loop for you.

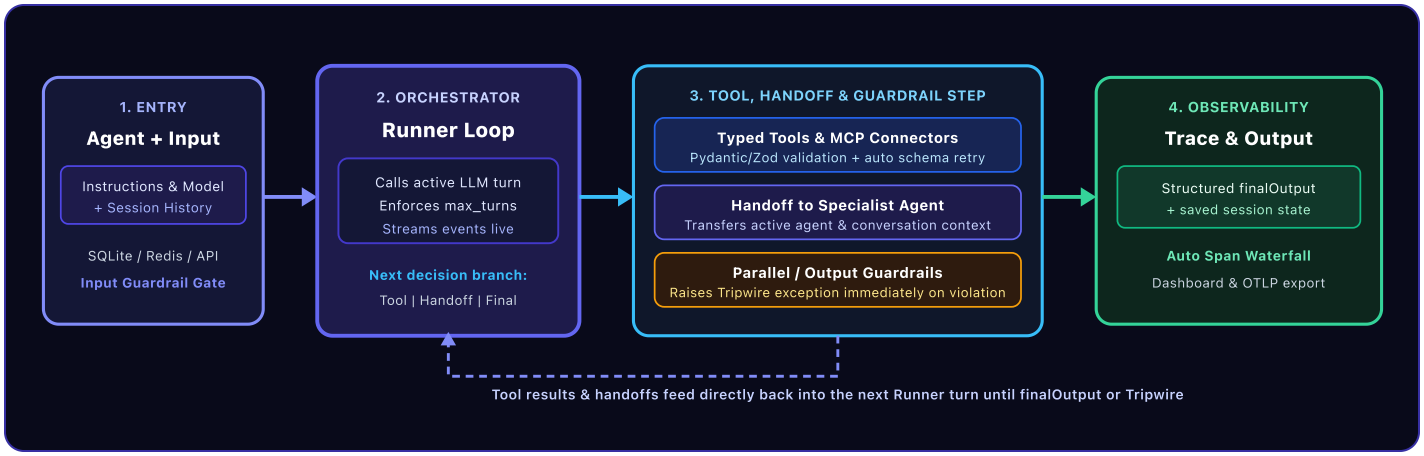
Quick Navigation

1. Core Loop & Mental Model SVG p. 2
2. Setup, Parity & API vs. SDK Matrix p. 3
3. Annotated Traces & Inspector Tour p. 4
4. Handoffs vs. Agents-as-Tools p. 5
5. Typed Tool Contracts & Schemas p. 6
6. Parallel Guardrails & Tracing p. 7
7. Durable Redis & MCP Server State p. 8
8. 5 Real-World Copy-Paste Recipes p. 9
9. Evals, Custom Exporters & Routing p. 10
10. Cross-Tool Synergy Pipeline p. 11
11. Troubleshooting Clinic & Checklist p. 12
Core Rule: Keep Specialists Focused All

From Swarm Experiments to a Production Harness

In early LLM apps, engineers wrote fragile `while True` loops by hand: parsing tool JSON, catching schema errors, appending messages, and guessing why an agent stalled. The **Agents SDK** replaces that plumbing with a transparent execution loop owned by `Runner`. You define lightweight `Agent` objects (instructions, model, tools, handoffs, guardrails) and hand them to `Runner.run()`. If you prefer zero infrastructure management, the hosted **Agents API** runs the exact same Codex harness server-side with durable Sessions (`v1/agents/sessions`), retries, and background execution.

Inside One `Runner` Execution Loop



Why Developers Love This Minimalist Design

Instead of learning proprietary graph DSLs, you write ordinary Python functions or TypeScript modules. Every step—tool calls, handoffs, MCP requests, and guardrail verdicts—emits structured spans automatically so debugging feels like reading a web request trace.

Installation, Language Parity & Decision Matrix

Both the Python and TypeScript SDKs share nearly identical concepts and defaults. Install your language runtime in one command, export `OPENAI_API_KEY`, and decide whether your loop runs in your own app process or inside OpenAI's managed Agents API.

1 Python SDK (`openai-agents`)

```
# Core SDK + Redis distributed session extra
pip install "openai-agents[redis]"

# Verify version & export credentials
export OPENAI_API_KEY="sk-..."
```

Requires Python 3.10+. Uses Pydantic v2 for strict tool schemas and async `asyncio` execution via `Runner.run()`.

2 TypeScript SDK (`@openai/agents`)

```
# Core SDK + Zod v3/v4 schema validation
npm install @openai/agents zod

# Works on Node 22+, Deno, Bun & Edge runtimes
export OPENAI_API_KEY="sk-..."
```

First-class TypeScript types infer `finalOutput` directly from your Zod schema with zero manual type casting.

Python vs. TypeScript Core API Parity

Capability	Python (`openai-agents`)	TypeScript (`@openai/agents`)
Define an Agent	`Agent(name=..., instructions=...)`	`new Agent({ name, instructions })`
Run Execution Loop	`await Runner.run(agent, prompt)`	`await run(agent, prompt)`
Typed Function Tool	`@function_tool` decorator + type hints	`tool({ parameters: z.object(...), execute })`
Agent Transfer / Tool	`handoff(specialist)` / `agent.as_tool(...)`	`handoff(specialist)` / `agent.asTool(...)`
Input / Output Guards	`InputGuardrail` / `OutputGuardrail`	`inputGuardrails` / `outputGuardrails`

Decision Matrix: Hosted Agents API vs. Self-Hosted Agents SDK

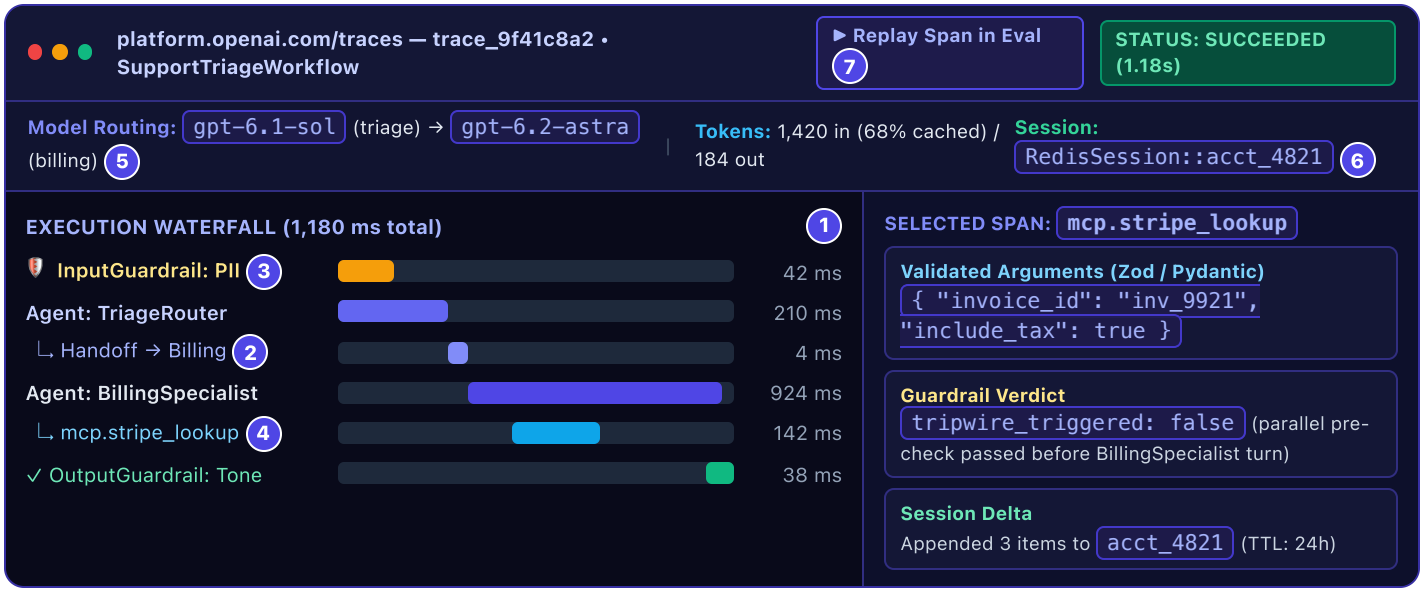
Dimension	Hosted OpenAI Agents API	Self-Hosted Agents SDK (Python / TS)
Execution & State	Managed server-side harness; OpenAI persists durable Sessions, sandbox environments, and compaction automatically.	Runs inside your own API server, worker queue, or CLI using `SQLiteSession` or `RedisSession`.
Custom Local Code	Connects via remote MCP servers, webhooks, or client-side tool callbacks.	Direct in-process access to private VPC databases, local files, internal SDKs, and custom middleware.
Data & Trace Control	Zero-maintenance OpenAI dashboard traces and conversation storage with enterprise retention settings.	Full control over memory TTLs, custom OTLP/Datadog trace exporters, and PII scrubbing before storage.
Best Fit	Async background jobs, web/mobile copilot backends, and teams minimizing DevOps overhead.	Latency-sensitive microservices, private VPC tooling, complex multi-model routing, and custom eval harnesses.

Hybrid Tip: Start in the SDK, Offload Long Runs to the Agents API

Because both share identical mental models—structured instructions, typed tools, and trace spans—you can prototype fast in local TypeScript or Python and move long-running asynchronous workflows to the hosted Agents API without rewriting your prompts.

Annotated Tour: OpenAI Traces & Run Inspector

Every run in the Agents SDK and Agents API emits a nested tree of spans automatically. Here are the seven diagnostic touchpoints inside the OpenAI Traces dashboard for spotting latency, handoff errors, and guardrail hits in seconds.



Inspector Legend: The 7 Trace Diagnostic Controls

1. **Run Waterfall Timeline:** Visualizes overlapping parallel work versus serial round-trips so you spot slow tools or extra model turns immediately.
2. **Handoff Transfer Span:** Records the exact moment one agent delegates control to a specialist, along with any filtered history passed forward.
3. **Parallel Guardrail Tripwire:** Shows input checks racing alongside the first turn and flags `tripwire_triggered: true` when blocked early.
4. **MCP & Function Tool Latency:** Separates external server/DB time (e.g., `142 ms`) from LLM inference time and captures schema validation retries.
5. **Model Tier & Cache Breakdown:** Tracks prompt-cache hit ratios and lets you verify cheap routing models handle triage before heavy reasoning runs.
6. **Session Memory Drawer:** Shows which `SQLiteSession`, `RedisSession`, or hosted thread ID read and persisted multi-turn state.
7. **One-Click Eval Replay:** Promotes any production trace—especially tripped guardrails or failed tool calls—directly into an OpenAI Evals regression test case.

Orchestration: Handoffs vs. Agents-as-Tools

Multi-agent engineering boils down to one architectural choice: should the specialist take over the conversation and speak directly to the user (**Handoff**), or perform a focused subtask and return data to a central manager (**Agent-as-Tool**)?

A Pattern 1: Specialist Handoffs

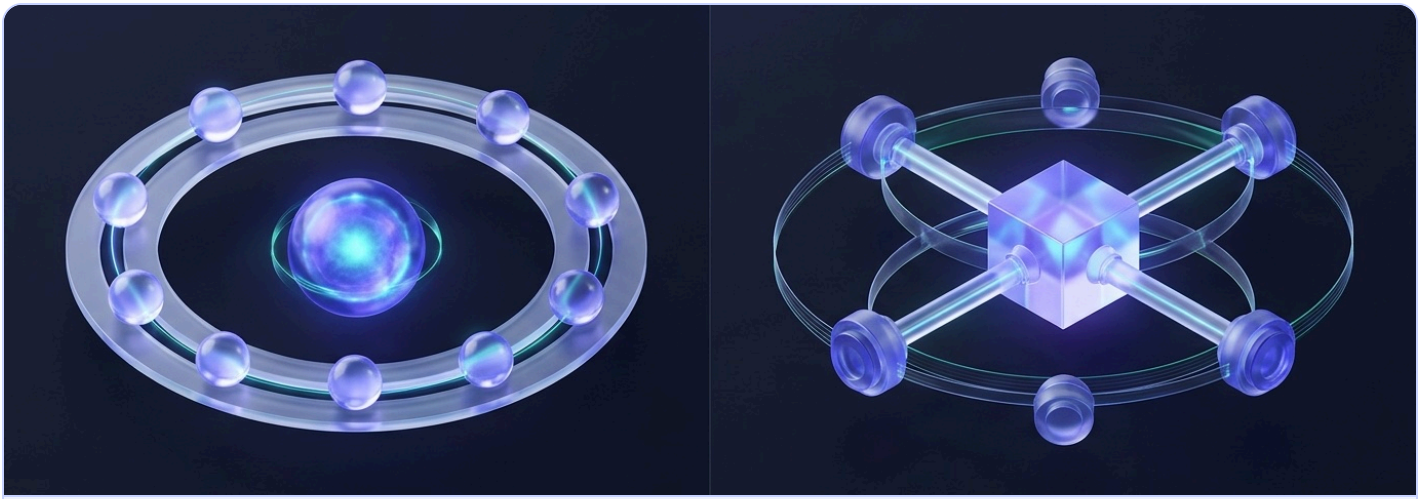
In a **Handoff**, the current agent invokes a built-in transfer tool (like `transfer_to_billing_agent`). The `Runner` swaps the active agent, passes conversation history (optionally pruned via `input_filter`), and lets the specialist own the user relationship from then on.

```
triage = Agent(
    name="TriageRouter",
    instructions="Route billing or tech questions.",
    handoffs=[billing_agent, tech_agent],
)
```

B Pattern 2: Agents-as-Tools

With `agent.as_tool()` (Python) or `agent.asTool()` (TypeScript), a sub-agent runs inside an isolated tool call and returns a string or JSON result back to the **Manager**. The manager keeps full ownership of tone, final synthesis, and multi-source aggregation.

```
manager = Agent(
    name="ReleaseManager",
    tools=[
        research_agent.as_tool(
            tool_name="fetch_changelog",
            tool_description="Summarize recent PRs."
        )
    ],
)
```



Left (Baton-Relay Handoff Ring): Control passes cleanly from node to node—ideal for customer triage, bilingual routing, and multi-stage onboarding where each specialist talks directly to the user. **Right (Hub-and-Spoke Manager):** A central conductor invokes specialist agents as tools in parallel and synthesizes one unified response.

Use Handoffs When:

- Different domains need distinct personas, languages, or toolsets.
- The user will continue chatting back and forth with that specialist.

Use Agents-as-Tools When:

- You need one consistent voice synthesizing multiple sub-tasks.
- Sub-agents should never leak intermediate reasoning to the user.

Structured Tool Contracts & Typed Outputs

Reliable agents never rely on regex parsing. The Agents SDK generates strict JSON Schemas straight from your Python type hints (Pydantic) or TypeScript Zod definitions—and automatically feeds validation errors back to the model for deterministic self-correction.

Python: `@function_tool` + Pydantic `output_type`

```python
from pydantic import BaseModel, Field
from agents import Agent, function_tool

class RefundDecision(BaseModel):
    approved: bool
    reason: str
    credit_cents: int = Field(ge=0)

@function_tool
def lookup_order(order_id: str) -> dict:
    """Fetch order status and purchase date by ID."""
    return db.get_order(order_id)

agent = Agent(
    name="RefundPolicy",
    tools=[lookup_order],
    output_type=RefundDecision,
)
```

TypeScript: `tool()` (`FunctionTool`) + Zod

```typescript
import { z } from "zod";
import { Agent, tool } from "@openai/agents";

const RefundDecision = z.object({
  approved: z.boolean(),
  reason: z.string(),
  creditCents: z.number().int().nonnegative(),
});

const lookupOrder = tool({
  name: "lookup_order",
  description: "Fetch order status and purchase date by
ID.",
  parameters: z.object({ orderId: z.string() }),
  execute: async ({ orderId }) => db.getOrder(orderId),
});

const agent = new Agent({
  name: "RefundPolicy",
  tools: [lookupOrder],
  outputType: RefundDecision,
});
```

System Prompt & Schema Anatomy: The 5 Color-Coded Building Blocks

Pair strict schemas with crisp instructions so the model knows *when* to invoke tools, *what* it must never invent, and *how* to populate your typed output:

[1. Agent Role] [2. Tool Call Rule] [3. Escalation Scope] [4. Deterministic Constraint] [5. Typed Output Contract]

You are the Senior Billing Policy Evaluator for Acme Cloud. Always call lookup_order(order_id) before evaluating eligibility; never guess invoice totals or purchase dates from memory. If the order involves enterprise annual contracts or fraud flags, hand off immediately to EnterpriseEscalation instead of issuing a decision. Approve automatic credits only for charges under 30 days old and under 10,000 cents; reject expired windows politely. Return your final verdict strictly matching the RefundDecision schema with non-negative integer credit_cents.

How Deterministic Schema Retries Save You From Fragile Parsing

If a model forgets a required argument or sends a string instead of an integer, the SDK intercepts the Pydantic or Zod `ValidationError`, returns the exact field error as a tool observation, and gives the model another turn to fix the payload—without crashing your API handler.

Defense-in-Depth Guardrails & Built-In Tracing

Treat prompts as guidance and **Guardrails** as enforceable gates. By running fast lightweight classifiers in parallel with your main agent, you block prompt injections, PII leaks, and off-topic abuse without adding extra serial latency.

1 Input Guardrails

Inspect user messages on entry. Default **parallel mode** starts checking while the main model begins thinking—canceling immediately if a tripwire fires, or set blocking mode to protect costly reasoning tokens.

2 Tool Guardrails

Validate arguments and side-effect scopes right before tool execution (e.g., blocking SQL writes, enforcing customer ID ownership, or masking secrets returned from MCP servers).

3 Output Guardrails

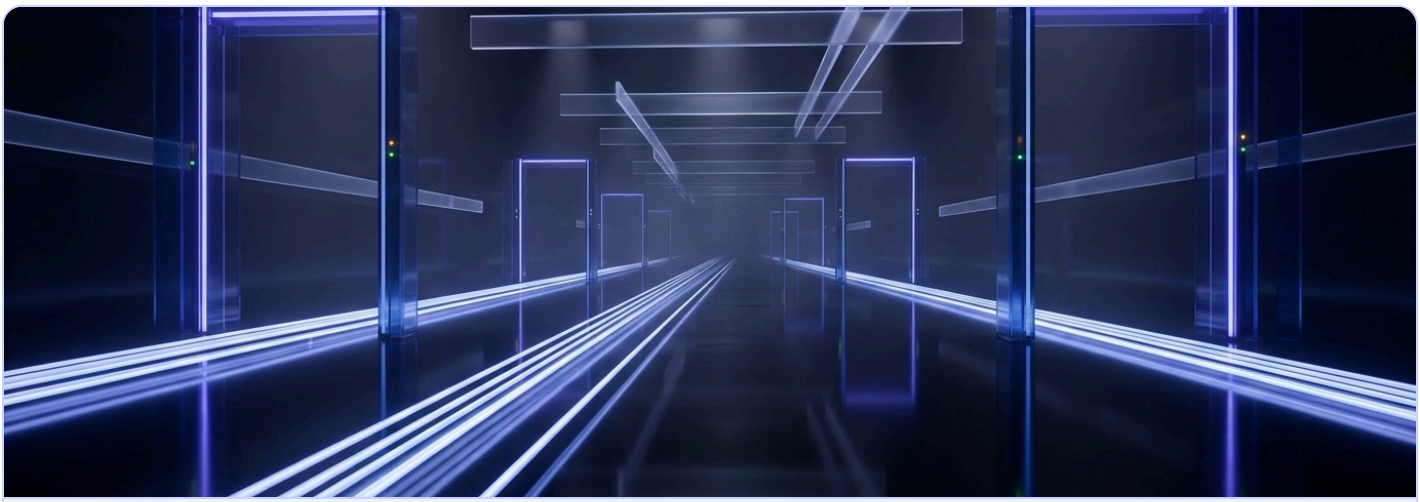
Verify the final response before it reaches the customer—checking for leaked system instructions, banned medical/legal claims, or unverified discount promises.

```
from agents import Agent, GuardrailFunctionOutput, InputGuardrail, InputGuardrailTripwireTriggered, Runner, trace

async def pii_gate(ctx, agent, user_input):
    verdict = await Runner.run(pii_classifier, user_input, context=ctx.context)
    return GuardrailFunctionOutput(
        output_info=verdict.final_output,
        tripwire_triggered=verdict.final_output.contains_ssn_or_card,
    )

support_agent = Agent(name="Support", input_guardrails=[InputGuardrail(guardrail_function=pii_gate)])

with trace("CustomerSupportTurn"):
    try:
        res = await Runner.run(support_agent, user_message)
    except InputGuardrailTripwireTriggered:
        res_text = "Please remove card numbers or SSNs before continuing."
```



Simultaneous Defense & Observability: Each stream passes through illuminated indigo input, tool, and output security gates while casting real-time span ribbons into the OpenAI Traces observatory—zero external APM agent required.

Group Multi-Turn Workflows With `with trace("WorkflowName")`

Wrapping several `Runner.run()` calls inside a single `trace()` block links triage, tool retries, and follow-up turns under one searchable parent timeline in the dashboard.

Durable State & Stateful MCP Server Sessions

Stateless scripts forget everything after one reply. The Agents SDK provides a drop-in `Session` protocol—starting with zero-setup `SQLiteSession` locally and scaling horizontally with `RedisSession`—alongside first-class connectors for persistent MCP tool servers.

1 Local Dev → Horizontal Redis

Pass a `session` directly to `Runner.run()`. The SDK automatically loads prior turns before inference and appends new tool and assistant items afterward.

```
# Local dev: zero external infra
from agents import SQLiteSession
session = SQLiteSession("user_42", "dev_chat.db")

# Production cluster: pip install "openai-agents[redis]"
from agents.extensions.memory import RedisSession
session = RedisSession.from_url(
    "user_42",
    url="redis://cache.internal:6379/0",
    ttl=86400,
)
result = await Runner.run(agent, msg, session=session)
```

2 Three Native MCP Transports

- **`MCPServerStdio`**: Launches local CLI tools (filesystem, git, sqlite) as a child process over stdin/stdout.
- **`MCPServerSse`**: Connects to legacy remote MCP servers using HTTP Server-Sent Events streams.
- **`MCPServerStreamableHttp`**: The modern production transport for cloud-hosted MCP gateways, supporting OAuth headers, clean bidirectional streaming, and load-balanced horizontal scaling.

Call `list_tools()` caching on connect to avoid redundant schema handshakes.

Wiring Persistent MCP Connectors Safely in Python & TypeScript

Always manage MCP server lifecycles with an async context manager (Python) or explicit `try / finally` cleanup (TypeScript) so child processes and HTTP keep-alive sockets close cleanly on errors:

```
# Python: Connect streamable HTTP + stdio MCP servers with guaranteed cleanup
from agents import Agent, Runner
from agents.mcp import MCPServerStdio, MCPServerStreamableHttp

async with (
    MCPServerStdio(params={"command": "npx", "args": ["-y", "@modelcontextprotocol/server-git"]}) as git_mcp,
    MCPServerStreamableHttp(params={"url": "https://mcp.internal.dev/mcp", "headers": {"Authorization": "Bearer ..."}}) as
prod_mcp,
):
    ops_agent = Agent(name="OpsResponder", mcp_servers=[git_mcp, prod_mcp], instructions="Diagnose incidents safely.")
    res = await Runner.run(ops_agent, "Check the last 5 commits and inspect failing service health.", session=session)
```

Tool Filtering Tip: Trim Unused MCP Tools Before Passing to the Agent

Large MCP servers can expose 40+ tools, inflating prompt tokens and confusing routing. Use `create_static_tool_filter(allowed_tool_names=[...])` or dynamic per-user filters on your MCP connector so each specialist sees only the 3–6 tools it truly needs.

5 Real-World Copy-Paste Recipes

Drop these five proven architectural patterns straight into your Python or TypeScript service. Each recipe pairs the exact SDK primitives needed for fast, deterministic execution.

Recipe Name	Primitives Used	Copy-Paste Implementation Blueprint

1. Triage Router with Bilingual Handoffs
Support Routing
	`Agent`
	`handoff()`
	`gpt-6.1-sol`
		Define `es_agent` and `en_agent` with localized instructions and tools. Attach `handoffs=[es_agent, en_agent]` to a lightweight `TriageRouter` running `gpt-6.1-sol` with instructions: *"Detect language on turn one and hand off immediately without answering."*

2. Manager Calling Research Agent as Tool
Executive Briefs
	`agent.as_tool()`
	`output_type`
	`asyncio.gather`
		Expose `web_researcher.as_tool("search_market", ...)` and `sec_analyst.as_tool("read_filings", ...)` on an orchestrator `BriefManager`. Enable parallel tool calls so both sub-agents execute simultaneously and return clean summaries for final Pydantic synthesis.

3. Parallel PII Input Guardrail Gate
Compliance Shield
	`InputGuardrail`
	`Tripwire`
	`run_in_parallel`
		Attach a fast classifier agent inside `InputGuardrail` checking for SSNs, API keys, or credit cards. Catch `InputGuardrailTripwireTriggered` around `Runner.run()` to return a polite redaction notice before any sensitive string enters downstream tool logs.

4. Persistent Redis Multi-Turn Concierge
Horizontally Scaled
	`RedisSession`
	`@function_tool`
	`ttl=86400`
		Initialize `RedisSession.from_url(f"chat:{tenant}:{user_id}", url=REDIS_URL, ttl=86400)` inside your FastAPI or Express handler and pass `session=session` to `Runner.run()`. Any stateless container pod resumes the exact multi-turn thread instantaneously.

5. MCP-Connected Incident Responder
Live SRE Triage
	`MCPServerStreamableHttp`
	`tool_filter`
	`trace()`
		Open read-only Sentry and GitHub MCP connectors with `allowed_tool_names=["get_issue", "get_diff"]` inside an `async with` block. Wrap `Runner.run(sre_agent, alert_payload)` in `with trace("IncidentTriage")` to capture root-cause hypotheses and span timings.

Composition Rule: Keep Every Agent Single-Purpose (<12 Tools)

When an agent collects more than 10–12 tools, tool selection accuracy drops. Split sprawling monoliths into focused specialist agents (Recipe 1 or 2), enforce boundaries with Recipe 3, and persist state cleanly with Recipe 4.

Production Ops, Evals & Cost Control

Shipping agents to production requires predictable latency, strict loop budgets, and continuous regression testing. The SDK exposes clean hooks for custom telemetry pipelines, eval datasets, and tiered model routing.

📡 Custom Trace Exporters & OTLP

By default, traces stream asynchronously to OpenAI's dashboard. Register a custom `TracingProcessor` (or `add_trace_processor`) to mirror spans into Datadog, Honeycomb, Grafana Tempo, or internal audit buckets without slowing user responses.

```
from agents.tracing import add_trace_processor,
BatchTraceProcessor
add_trace_processor(BatchTraceProcessor(OtlpSpanExporter()))
```

2 Eval Replay From Failed Spans

Never fix a production agent bug by guesswork. Export failed or tripped trace spans directly into an OpenAI Evals suite (or pytest/Vitest fixture) so every prompt or schema tweak is verified against real edge cases before deploy.

- Assert exact tool-call sequence and handoff destination.
- Grade final structured outputs with deterministic schema + LLM-as-judge checks.

Execution Budgets & Dynamic Model Tier Routing

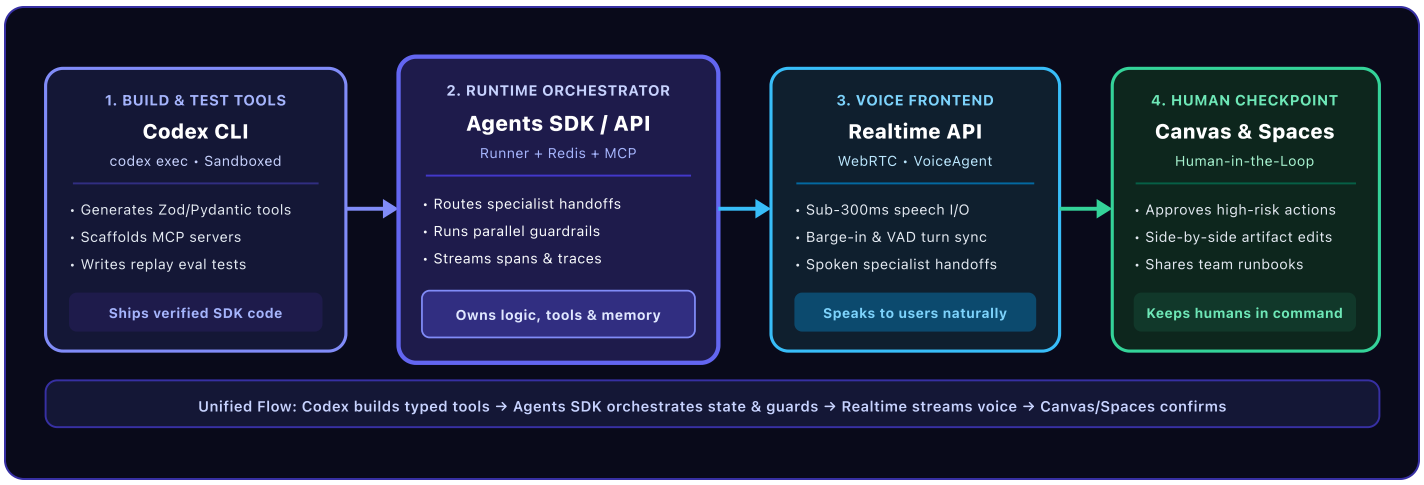
Control Lever	SDK Configuration	Production Impact
Loop Circuit Breaker	`Runner.run(agent, msg, max_turns=8)`	Raises `MaxTurnsExceeded` cleanly instead of burning unlimited tokens in a circular tool or handoff loop.
Fast Triage & Guardrails	`model="gpt-6.1-sol"`	Handles input/output guardrail checks, intent routing, and simple JSON extraction in ~100–250 ms at a fraction of flagship cost.
Flagship Reasoning Tier	`model="gpt-6.2-astra"` / `gpt-5.4`	Reserved strictly for complex specialist synthesis, multi-step code diagnostics, and high-stakes financial/policy planning.
Sensitive Data Scrubbing	`RunConfig(trace_include_sensitive_data=False)`	Records span timings, token counts, and tool names while stripping raw prompt/tool payloads in regulated HIPAA/PCI environments.

Dynamic Routing Rule: Pair `gpt-6.1-sol` Routers With Flagship Specialists

Running a single flagship model for guardrails, routing, and formatting wastes up to 70% of spend and doubles time-to-first-token. Put `gpt-6.1-sol` on routers, guardrails, and lightweight tools, and hand off only genuinely hard reasoning turns to flagship models.

Cross-Tool Synergy: Full OpenAI Agent Stack

The Agents API and SDK sit at the center of OpenAI's builder ecosystem—connecting developer-built tools from **Codex CLI**, sub-second **Realtime Voice** interfaces, and visual **Canvas / Spaces** human-in-the-loop checkpoints into one cohesive architecture.



How These Four Layers Cooperate in Practice

1 Build With Codex, Orchestrate in the SDK

Use **Codex CLI** to scaffold strictly typed `@function_tool` handlers, Zod schemas, and unit-tested MCP adapters. Your **Agents SDK** backend mounts those tools behind parallel input guardrails and durable `RedisSession` history.

2 Voice Speed + Human Approval Gates

Pair a `RealtimeSession` voice frontend for instant spoken triage while delegating deep database or billing mutations to your Agents SDK backend—pausing for one-click human approval inside **Canvas** or **Spaces** before irreversible writes.

One Trace ID Across Voice, Backend Agents & UI

Propagate a single `trace_id` from the Realtime voice client through your Agents SDK backend and MCP calls so product engineers can replay an entire multi-modal customer conversation from one unified timeline.

Troubleshooting Clinic & Pre-Flight Checklist

Almost every multi-agent bug traces back to one of seven well-known runtime traps. Use this quick clinic to diagnose failures in ten seconds, then clear the six-point pre-flight checklist before shipping to production.

Symptom / Error	Root Cause	10-Second Fix
1. Infinite handoff ping-pong between two agents	Both specialists include each other in `handoffs=[...]` with overlapping routing instructions.	Make handoffs a clean DAG (Router → Specialist), clarify negative routing boundaries, and cap turns with `max_turns=8`.
2. Process crashes with `GuardrailTripwireTriggered`	`Runner.run()` raised a tripwire exception that wasn't caught by the API route handler.	Wrap `Runner.run()` in `try / except (InputGuardrailTripwireTriggered, OutputGuardrailTripwireTriggered)` and return a safe HTTP 400/200 fallback.
3. Strict JSON schema error on tool / `output_type`	OpenAI strict mode rejects optional fields without defaults, untyped `dict` parameters, or `additionalProperties`.	Use explicit Pydantic models or Zod objects with concrete types, clear field descriptions, and no bare unannotated dictionaries.
4. `RedisSession` stalls or times out under load	A new Redis TCP client is created per request instead of sharing an async connection pool.	Instantiate one shared `redis.asyncio` connection pool at app startup and pass it into `RedisSession` with an explicit `ttl`.
5. Zombie MCP subprocesses or socket leaks	`MCPServerStdio` or `MCPServerStreamableHttp` was opened manually without closing on error.	Always enter MCP servers via `async with server:` (Python) or `try { ... } finally { await server.close(); }` (TypeScript).
6. Customer PII appearing in OpenAI trace payloads	Default tracing logs raw prompt and tool arguments when compliance requires metadata-only telemetry.	Pass `RunConfig(trace_include_sensitive_data=False)` or scrub fields inside a custom `TracingProcessor` before export.
7. `MaxTurnsExceeded` aborts multi-step workflow	An agent hits the default turn cap while sequentially polling 10+ tool calls one by one.	Enable parallel tool calling, batch lookups into a single tool, or raise `max_turns` intentionally for deep research loops.

6-Point Pre-Flight Checklist Before Production Deploy

- ☑ **Loop Budget Set:** Every `Runner.run()` call sets an explicit `max_turns` cap and handles `MaxTurnsExceeded`.
- ☑ **Tripwires Caught Cleanly:** Both input and output guardrail exceptions return user-friendly fallback replies.
- ☑ **Strict Schemas Verified:** All tools and `output_type` classes pass Pydantic/Zod strict JSON Schema checks.
- ☑ **Durable Session Pooled:** Replaced local `SQLiteSession` with pooled `RedisSession` (plus TTL) or hosted Agents API Sessions.
- ☑ **MCP Lifecycles Scoped:** Connected MCP servers run inside context managers with filtered tool allowlists.
- ☑ **Traces & Evals Wired:** `trace()` spans scrub sensitive data where needed and failed spans feed your eval suite.