Building Consistent Workflows with Codex CLI & Agents SDK

Ensuring Repeatable, Traceable, and Scaleable Agentic Development

Introduction

Developers strive for consistency in everything they do. With Codex CLI and the Agents SDK, that consistency can now scale like never before. Whether you're refactoring a large codebase, rolling out new features, or introducing a new testing framework, Codex integrates seamlessly into CLI, IDE, and cloud workflows to automate and enforce repeatable development patterns.

In this track, we'll build both single and multi-agent systems using the Agents SDK, with Codex CLI exposed as an MCP Server. This enables:
- **Consistency and Repeatability** by providing each agent a scoped context.
- **Scalable Orchestration** to coordinate single and multi-agent systems.
- **Observability & Auditability** by reviewing the full agentic stack trace.

What We'll Cover

- Initializing Codex CLI as an MCP Server: How to run Codex as a long-running MCP process.
- Building Single-Agent Systems: Using Codex MCP for scoped tasks.
- Orchestrating Multi-Agent Workflows: Coordinating multiple specialized agents.
- Tracing Agentic Behavior: Leveraging agent traces for visibility and evaluation.

Prerequisites & Setup

Before starting this track, ensure you have the following:
- Basic coding familiarity: You should be comfortable with Python and JavaScript.
- Developer environment: You'll need an IDE, like VS Code or Cursor.
- OpenAI API key: Create or find your API key in the OpenAI Dashboard.

Environment Setup

1. create a `.env` folder in your directory and add your `OPENAI_API_KEY` Key
2. Install dependencies

```
%pip install openai-agents openai ## install dependencies
```

Initializing Codex CLI as an MCP Server

Here run Codex CLI as an MCP Server inside the Agents SDK. We provide the initialization parameters of `codex mcp`. This command starts Codex CLI as an MCP server and exposes two Codex tools available on the MCP server — `codex()` and `codex-reply()`. These are the underlying tools that the Agents SDK will call when it needs to invoke Codex.
- `codex()` is used for creating a conversation.
- `codex-reply()` is for continuing a conversation.

```
import asyncio
from agents import Agent, Runner
from agents.mcp import MCPServerStdio

async def main() -> None:
    async with MCPServerStdio(
        name="Codex CLI",
        params={
            "command": "npx",
            "args": ["-y", "codex", "mcp-server"],
        },
        client_session_timeout_seconds=360000,
    ) as codex_mcp_server:
        print("Codex MCP server started.")
        # We will add more code here in the next section
        return
```

Also note that we are extending the MCP Server timeout to allow Codex CLI enough time to execute and complete the given task.

Building Single Agent Systems

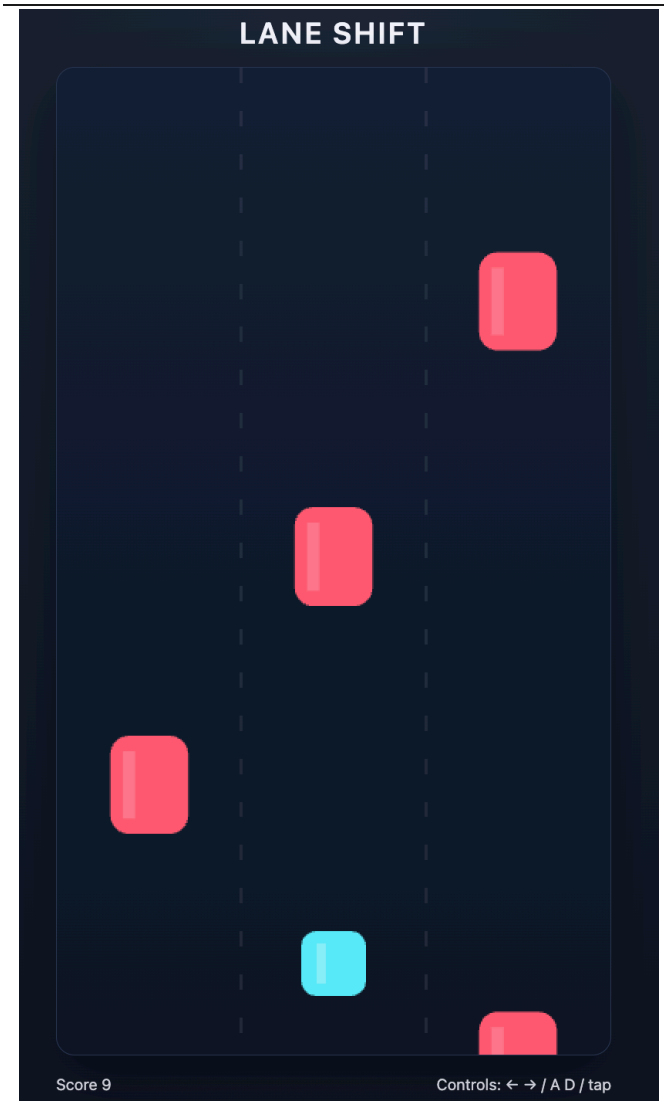
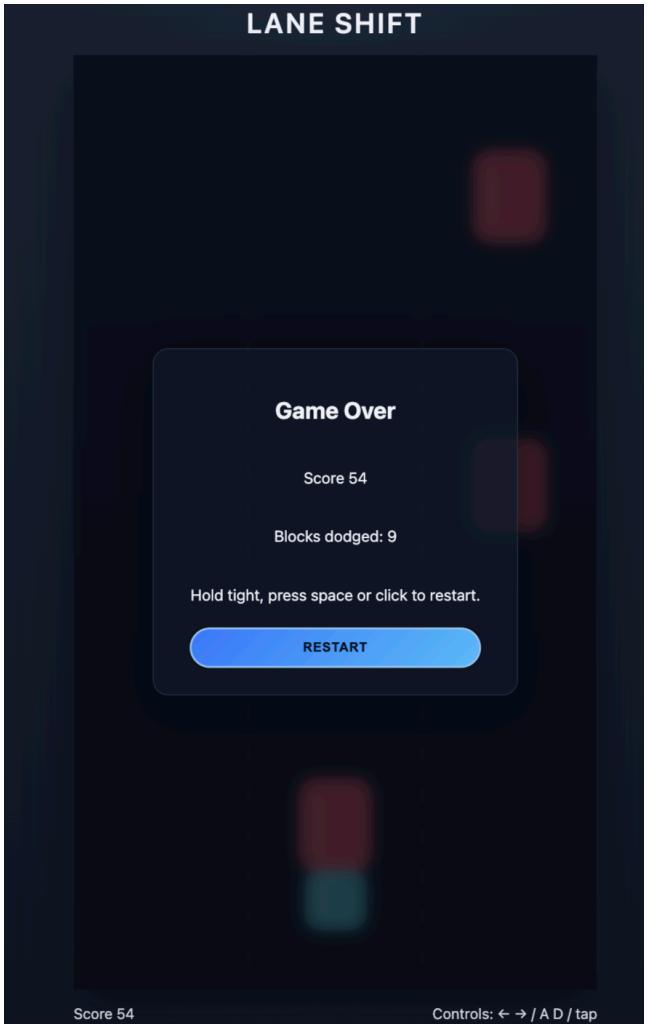
Let's start with a simple example to use our Codex MCP Server. We define two agents:
1. **Designer Agent** – brainstorms and creates a small brief for a game.
2. **Developer Agent** – implements a simple game according to the Designer's spec.

```
developer_agent = Agent(
    name="Game Developer",
    instructions=(
        "You are an expert in building simple games using basic html
        + css + javascript with no dependencies. "
        "Save your work in a file called index.html in the current
        directory."
        "Always call codex with \"approval-policy\": \"never\" and
        \"sandbox\": \"workspace-write\""
    ),
    mcp_servers=[codex_mcp_server],
)

designer_agent = Agent(
    name="Game Designer",
    instructions=(
        "You are an indie game connoisseur. Come up with an idea for
        a single page html + css + javascript game that a developer
        could build in about 50 lines of code. "
        "Format your request as a 3 sentence design brief for a game
        developer and call the Game Developer coder with your idea."
    ),
    model="gpt-5",
    handoffs=[developer_agent],
)

result = await Runner.run(designer_agent, "Implement a fun new
        game!")
```

Notice that we are providing the Developer agent with the ability to write files to the project directory without asking the user for permissions.

Now run the code and you'll see an `index.html` file generated. Go ahead and open the file and start playing the game!

Here's a few screenshots of the game my agentic system created. Yours will be different!

Example gameplay	Game Over Score
	

Here's the full executable code. Note that it might take a few minutes to run. It will have run successfully if you see an index.html file produced. You might also see some MCP events warnings about format. You can ignore these events.

```
import os
from dotenv import load_dotenv
import asyncio
from agents import Agent, Runner, set_default_openai_api
from agents.mcp import MCPServerStdio

load_dotenv(override=True) # load the API key from the .env file. We
        set override to True here to ensure the notebook is loading
        any changes
```

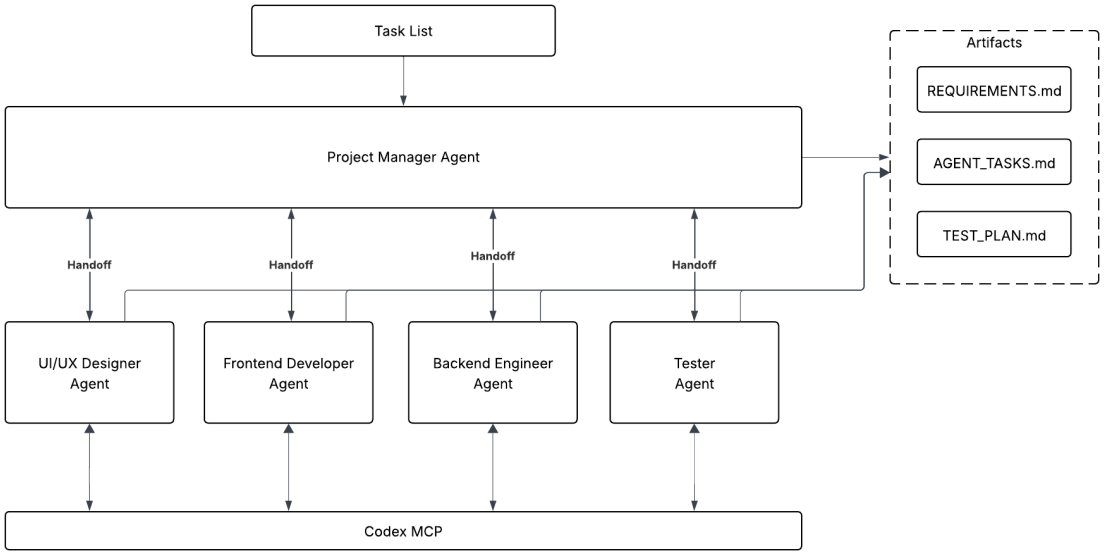
```python
set_default_openai_api(os.getenv("OPENAI_API_KEY"))

async def main() -> None:
    async with MCPServerStdio(
        name="Codex CLI",
        params={
            "command": "npx",
            "args": ["-y", "codex", "mcp-server"],
        },
        client_session_timeout_seconds=360000,
    ) as codex_mcp_server:
        developer_agent = Agent(
            name="Game Developer",
            instructions=(
                "You are an expert in building simple games using basic html + css + javascript with no dependencies. "
                "Save your work in a file called index.html in the current directory."
                "Always call codex with \"approval-policy\": \"never\" and \"sandbox\": \"workspace-write\""
            ),
            mcp_servers=[codex_mcp_server],
        )

        designer_agent = Agent(
            name="Game Designer",
            instructions=(
                "You are an indie game connoisseur. Come up with an idea for a single page html + css + javascript game that a developer could build in about 50 lines of code. "
                "Format your request as a 3 sentence design brief for a game developer and call the Game Developer coder with your idea."
            ),
            model="gpt-5",
            handoffs=[developer_agent],
        )

        result = await Runner.run(designer_agent, "Implement a fun new game!")
        # print(result.final_output)


if __name__ == "__main__":
    # Jupyter/IPython already runs an event loop, so calling asyncio.run() here
```

```
    # raises "asyncio.run() cannot be called from a running event
        loop".
    # Workaround: if a loop is running (notebook), use top-level
        `await`; otherwise use asyncio.run().
    try:
        asyncio.get_running_loop()
        await main()
    except RuntimeError:
        asyncio.run(main())
```

Orchestrating Multi-Agent Workflows

For larger workflows, we introduce a team of agents:
- **Project Manager**: Breaks down task list, creates requirements, and coordinates work.
- **Designer**: Produces UI/UX specifications.
- **Frontend Developer**: Implements UI/UX.
- **Backend Developer**: Implements APIs and logic.
- **Tester**: Validates outputs against acceptance criteria.

In this example, we intentionally have the Project Manager agent enforce gating logic between each of the specialized downstream agents. This ensures that artifacts exist before handoffs are made. This mirrors real world enterprise workflows such as JIRA task orchestration, long-chained rollouts, and QA sign-offs.



Multi-agent orchestration with Codex MCP and gated handoffs producing artifacts.

In this structure, each of our agents serve a specialized purpose. The Project Manager is overall responsible for coordinating across all other agents and ensuring the overall task is complete.

Define the Codex CLI MCP Server

We set up our MCP Server to initialize Codex CLI just as we did in the single agent example.

```
async def main() -> None:
    async with MCPServerStdio(
        name="Codex CLI",
        params={
            "command": "npx",
            "args": ["-y", "codex", "mcp-server"],
        },
        client_session_timeout_seconds=360000,
    ) as codex_mcp_server:
        print("Codex MCP server started.")
        # We will add more code here in the next section
        return
```

Define each specialized agent

Below we define each of our specialized agents and provide access to our Codex MCP server. Notice that we are also passing the `RECOMMMENDED_PROMPT_PREFIX` to each agent that helps the system optimize for handoffs between agents.

```
# Downstream agents are defined first for clarity, then PM references
        them in handoffs.
designer_agent = Agent(
    name="Designer",
    instructions=(
        f"""{RECOMMENDED_PROMPT_PREFIX}"""
        "You are the Designer.\n"
        "Your only source of truth is AGENT_TASKS.md and
        REQUIREMENTS.md from the Project Manager.\n"
        "Do not assume anything that is not written there.\n\n"
        "You may use the internet for additional guidance or
        research."
        "Deliverables (write to /design):\n"
        "- design_spec.md - a single page describing the UI/UX
        layout, main screens, and key visual notes as requested in
        AGENT_TASKS.md.\n"
```

```
            "- wireframe.md - a simple text or ASCII wireframe if
            specified.\n\n"
            "Keep the output short and implementation-friendly.\n"
            "When complete, handoff to the Project Manager with
            transfer_to_project_manager."
            "When creating files, call Codex MCP with {\"approval-
            policy\":\"never\",\"sandbox\":\"workspace-write\"}."
        ),
        model="gpt-5",
        tools=[WebSearchTool()],
        mcp_servers=[codex_mcp_server],
        handoffs=[],
    )

    frontend_developer_agent = Agent(
        name="Frontend Developer",
        instructions=(
            f"""{RECOMMENDED_PROMPT_PREFIX}"""
            "You are the Frontend Developer.\n"
            "Read AGENT_TASKS.md and design_spec.md. Implement exactly
            what is described there.\n\n"
            "Deliverables (write to /frontend):\n"
            "- index.html - main page structure\n"
            "- styles.css or inline styles if specified\n"
            "- main.js or game.js if specified\n\n"
            "Follow the Designer's DOM structure and any integration
            points given by the Project Manager.\n"
            "Do not add features or branding beyond the provided
            documents.\n\n"
            "When complete, handoff to the Project Manager with
            transfer_to_project_manager_agent."
            "When creating files, call Codex MCP with {\"approval-
            policy\":\"never\",\"sandbox\":\"workspace-write\"}."
        ),
        model="gpt-5",
        mcp_servers=[codex_mcp_server],
        handoffs=[],
    )

    backend_developer_agent = Agent(
        name="Backend Developer",
        instructions=(
            f"""{RECOMMENDED_PROMPT_PREFIX}"""
            "You are the Backend Developer.\n"
            "Read AGENT_TASKS.md and REQUIREMENTS.md. Implement the
            backend endpoints described there.\n\n"
            "Deliverables (write to /backend):\n"
```

```
            "- package.json - include a start script if requested\n"
            "- server.js - implement the API endpoints and logic exactly
            as specified\n\n"
            "Keep the code as simple and readable as possible. No
            external database.\n\n"
            "When complete, handoff to the Project Manager with
            transfer_to_project_manager_agent."
            "When creating files, call Codex MCP with {\"approval-
            policy\":\"never\",\"sandbox\":\"workspace-write\"}."
        ),
        model="gpt-5",
        mcp_servers=[codex_mcp_server],
        handoffs=[],
    )

    tester_agent = Agent(
        name="Tester",
        instructions=(
            f"""{RECOMMENDED_PROMPT_PREFIX}"""
            "You are the Tester.\n"
            "Read AGENT_TASKS.md and TEST.md. Verify that the outputs of
            the other roles meet the acceptance criteria.\n\n"
            "Deliverables (write to /tests):\n"
            "- TEST_PLAN.md - bullet list of manual checks or automated
            steps as requested\n"
            "- test.sh or a simple automated script if specified\n\n"
            "Keep it minimal and easy to run.\n\n"
            "When complete, handoff to the Project Manager with
            transfer_to_project_manager."
            "When creating files, call Codex MCP with {\"approval-
            policy\":\"never\",\"sandbox\":\"workspace-write\"}."
        ),
        model="gpt-5",
        mcp_servers=[codex_mcp_server],
        handoffs=[],
    )
```

After each role completes its assignment, it will call `transfer_to_project_manager_agent`, and let the Project Manager confirm that the required files exist (or request fixes) before unblocking the next team.

Define Project Manager Agent

The Project Manager is the only agent that receives the initial prompt, creates the planning documents in the project directory, and enforces the gatekeeping logic before every transfer.

```
project_manager_agent = Agent(
name="Project Manager",
instructions=(
    f"""{RECOMMENDED_PROMPT_PREFIX}"""
    """
    You are the Project Manager.

    Objective:
    Convert the input task list into three project-root files the team
        will execute against.

    Deliverables (write in project root):
    - REQUIREMENTS.md: concise summary of product goals, target
        users, key features, and constraints.
    - TEST.md: tasks with [Owner] tags (Designer, Frontend, Backend,
        Tester) and clear acceptance criteria.
    - AGENT_TASKS.md: one section per role containing:
        - Project name
        - Required deliverables (exact file names and purpose)
        - Key technical notes and constraints

    Process:
    - Resolve ambiguities with minimal, reasonable assumptions. Be
        specific so each role can act without guessing.
    - Create files using Codex MCP with {"approval-
        policy":"never","sandbox":"workspace-write"}.
    - Do not create folders. Only create REQUIREMENTS.md, TEST.md,
        AGENT_TASKS.md.

    Handoffs (gated by required files):
    1) After the three files above are created, hand off to the
        Designer with transfer_to_designer_agent and include
        REQUIREMENTS.md, and AGENT_TASKS.md.
    2) Wait for the Designer to produce /design/design_spec.md.
        Verify that file exists before proceeding.
    3) When design_spec.md exists, hand off in parallel to both:
        - Frontend Developer with
        transfer_to_frontend_developer_agent (provide design_spec.md,
        REQUIREMENTS.md, AGENT_TASKS.md).
```

```
        - Backend Developer with transfer_to_backend_developer_agent
          (provide REQUIREMENTS.md, AGENT_TASKS.md).
    4) Wait for Frontend to produce /frontend/index.html and Backend
        to produce /backend/server.js. Verify both files exist.
    5) When both exist, hand off to the Tester with
        transfer_to_tester_agent and provide all prior artifacts and
        outputs.
    6) Do not advance to the next handoff until the required files for
        that step are present. If something is missing, request the
        owning agent to supply it and re-check.

    PM Responsibilities:
    - Coordinate all roles, track file completion, and enforce the
        above gating checks.
    - Do NOT respond with status updates. Just handoff to the next
        agent until the project is complete.
    """
),
model="gpt-5",
model_settings=ModelSettings(
    reasoning=Reasoning(effort="medium")
),
handoffs=[designer_agent, frontend_developer_agent,
        backend_developer_agent, tester_agent],
mcp_servers=[codex_mcp_server],
)
```

After constructing the Project Manager, the script sets every specialist's handoffs back to the Project Manager. This ensures deliverables return for validation before moving on.

```
designer_agent.handoffs = [project_manager_agent]
frontend_developer_agent.handoffs = [project_manager_agent]
backend_developer_agent.handoffs = [project_manager_agent]
tester_agent.handoffs = [project_manager_agent]
```

Add in your task list

This is the task that the Project Manager will refine into specific requirements and tasks for the entire system.

```
task_list = """
Goal: Build a tiny browser game to showcase a multi-agent workflow.

High-level requirements:
- Single-screen game called "Bug Busters".
- Player clicks a moving bug to earn points.
```

```
- Game ends after 20 seconds and shows final score.
- Optional: submit score to a simple backend and display a top-10
        leaderboard.

Roles:
- Designer: create a one-page UI/UX spec and basic wireframe.
- Frontend Developer: implement the page and game logic.
- Backend Developer: implement a minimal API (GET /health, GET/POST
        /scores).
- Tester: write a quick test plan and a simple script to verify core
        routes.

Constraints:
- No external database—memory storage is fine.
- Keep everything readable for beginners; no frameworks required.
- All outputs should be small files saved in clearly named folders.
"""
```

Next, run your system, sit back, and you'll see the agents go to work and create a game in a few minutes! We've included the fully executable code below. Once it's finished, you'll notice the creation of the following files directory. Note that this multi-agent orchestration usually took about 11 mintues to fully complete.

```
root_directory/
├── AGENT_TASKS.md
├── REQUIREMENTS.md
├── backend
│   ├── package.json
│   └── server.js
├── design
│   ├── design_spec.md
│   └── wireframe.md
├── frontend
│   ├── game.js
│   ├── index.html
│   └── styles.css
└── TEST.md
```

Start your backend server with `node server.js` and open your `index.html` file to play your game.

```
import os
from dotenv import load_dotenv
import asyncio
from agents import Agent, Runner, WebSearchTool, ModelSettings,
        set_default_openai_api
from agents.mcp import MCPServerStdio
```

```
from agents.extensions.handoff_prompt import
        RECOMMENDED_PROMPT_PREFIX
from openai.types.shared import Reasoning

load_dotenv(override=True) # load the API key from the .env file. We
        set override to True here to ensure the notebook is loading
        any changes
set_default_openai_api(os.getenv("OPENAI_API_KEY"))

async def main() -> None:
    async with MCPServerStdio(
        name="Codex CLI",
        params={"command": "npx", "args": ["-y", "codex", "mcp-
        server"]},
        client_session_timeout_seconds=360000,
    ) as codex_mcp_server:

        # Downstream agents are defined first for clarity, then PM
        references them in handoffs.
        designer_agent = Agent(
            name="Designer",
            instructions=(
                f"""{RECOMMENDED_PROMPT_PREFIX}"""
                "You are the Designer.\n"
                "Your only source of truth is AGENT_TASKS.md and
        REQUIREMENTS.md from the Project Manager.\n"
                "Do not assume anything that is not written
        there.\n\n"
                "You may use the internet for additional guidance or
        research."
                "Deliverables (write to /design):\n"
                "- design_spec.md - a single page describing the UI/
        UX layout, main screens, and key visual notes as requested in
        AGENT_TASKS.md.\n"
                "- wireframe.md - a simple text or ASCII wireframe if
        specified.\n\n"
                "Keep the output short and implementation-
        friendly.\n"
                "When complete, handoff to the Project Manager with
        transfer_to_project_manager."
                "When creating files, call Codex MCP with {\"approval-
        policy\":\"never\",\"sandbox\":\"workspace-write\"}."
            ),
            model="gpt-5",
            tools=[WebSearchTool()],
            mcp_servers=[codex_mcp_server],
            handoffs=[],
```

```
)

frontend_developer_agent = Agent(
    name="Frontend Developer",
    instructions=(
        f"""{RECOMMENDED_PROMPT_PREFIX}"""
        "You are the Frontend Developer.\n"
        "Read AGENT_TASKS.md and design_spec.md. Implement
exactly what is described there.\n\n"
        "Deliverables (write to /frontend):\n"
        "- index.html — main page structure\n"
        "- styles.css or inline styles if specified\n"
        "- main.js or game.js if specified\n\n"
        "Follow the Designer's DOM structure and any
integration points given by the Project Manager.\n"
        "Do not add features or branding beyond the provided
documents.\n\n"
        "When complete, handoff to the Project Manager with
transfer_to_project_manager_agent."
        "When creating files, call Codex MCP with {\"approval-
policy\":\"never\",\"sandbox\":\"workspace-write\"}."
    ),
    model="gpt-5",
    mcp_servers=[codex_mcp_server],
    handoffs=[],
)

backend_developer_agent = Agent(
    name="Backend Developer",
    instructions=(
        f"""{RECOMMENDED_PROMPT_PREFIX}"""
        "You are the Backend Developer.\n"
        "Read AGENT_TASKS.md and REQUIREMENTS.md. Implement
the backend endpoints described there.\n\n"
        "Deliverables (write to /backend):\n"
        "- package.json — include a start script if
requested\n"
        "- server.js — implement the API endpoints and logic
exactly as specified\n\n"
        "Keep the code as simple and readable as possible. No
external database.\n\n"
        "When complete, handoff to the Project Manager with
transfer_to_project_manager_agent."
        "When creating files, call Codex MCP with {\"approval-
policy\":\"never\",\"sandbox\":\"workspace-write\"}."
    ),
    model="gpt-5",
```

```
    mcp_servers=[codex_mcp_server],
    handoffs=[],
)

tester_agent = Agent(
    name="Tester",
    instructions=(
        f"""{RECOMMENDED_PROMPT_PREFIX}"""
        "You are the Tester.\n"
        "Read AGENT_TASKS.md and TEST.md. Verify that the
outputs of the other roles meet the acceptance criteria.\n\n"
        "Deliverables (write to /tests):\n"
        "- TEST_PLAN.md — bullet list of manual checks or
automated steps as requested\n"
        "- test.sh or a simple automated script if
specified\n\n"
        "Keep it minimal and easy to run.\n\n"
        "When complete, handoff to the Project Manager with
transfer_to_project_manager."
        "When creating files, call Codex MCP with {\"approval-
policy\":\"never\",\"sandbox\":\"workspace-write\"}."
    ),
    model="gpt-5",
    mcp_servers=[codex_mcp_server],
    handoffs=[],
)

project_manager_agent = Agent(
    name="Project Manager",
    instructions=(
        f"""{RECOMMENDED_PROMPT_PREFIX}"""
        """
        You are the Project Manager.

        Objective:
        Convert the input task list into three project-root
files the team will execute against.

        Deliverables (write in project root):
        - REQUIREMENTS.md: concise summary of product goals,
target users, key features, and constraints.
        - TEST.md: tasks with [Owner] tags (Designer,
Frontend, Backend, Tester) and clear acceptance criteria.
        - AGENT_TASKS.md: one section per role containing:
          - Project name
          - Required deliverables (exact file names and
purpose)
```

```
          - Key technical notes and constraints

        Process:
        - Resolve ambiguities with minimal, reasonable
assumptions. Be specific so each role can act without
guessing.
        - Create files using Codex MCP with {"approval-
policy":"never","sandbox":"workspace-write"}.
        - Do not create folders. Only create REQUIREMENTS.md,
TEST.md, AGENT_TASKS.md.

        Handoffs (gated by required files):
        1) After the three files above are created, hand off
to the Designer with transfer_to_designer_agent and include
REQUIREMENTS.md, and AGENT_TASKS.md.
        2) Wait for the Designer to produce /design/
design_spec.md. Verify that file exists before proceeding.
        3) When design_spec.md exists, hand off in parallel
to both:
           - Frontend Developer with
transfer_to_frontend_developer_agent (provide design_spec.md,
REQUIREMENTS.md, AGENT_TASKS.md).
           - Backend Developer with
transfer_to_backend_developer_agent (provide REQUIREMENTS.md,
AGENT_TASKS.md).
        4) Wait for Frontend to produce /frontend/index.html
and Backend to produce /backend/server.js. Verify both files
exist.
        5) When both exist, hand off to the Tester with
transfer_to_tester_agent and provide all prior artifacts and
outputs.
        6) Do not advance to the next handoff until the
required files for that step are present. If something is
missing, request the owning agent to supply it and re-check.

        PM Responsibilities:
        - Coordinate all roles, track file completion, and
enforce the above gating checks.
        - Do NOT respond with status updates. Just handoff to
the next agent until the project is complete.
        """
    ),
    model="gpt-5",
    model_settings=ModelSettings(
        reasoning=Reasoning(effort="medium")
    ),
    handoffs=[designer_agent, frontend_developer_agent,
backend_developer_agent, tester_agent],
```

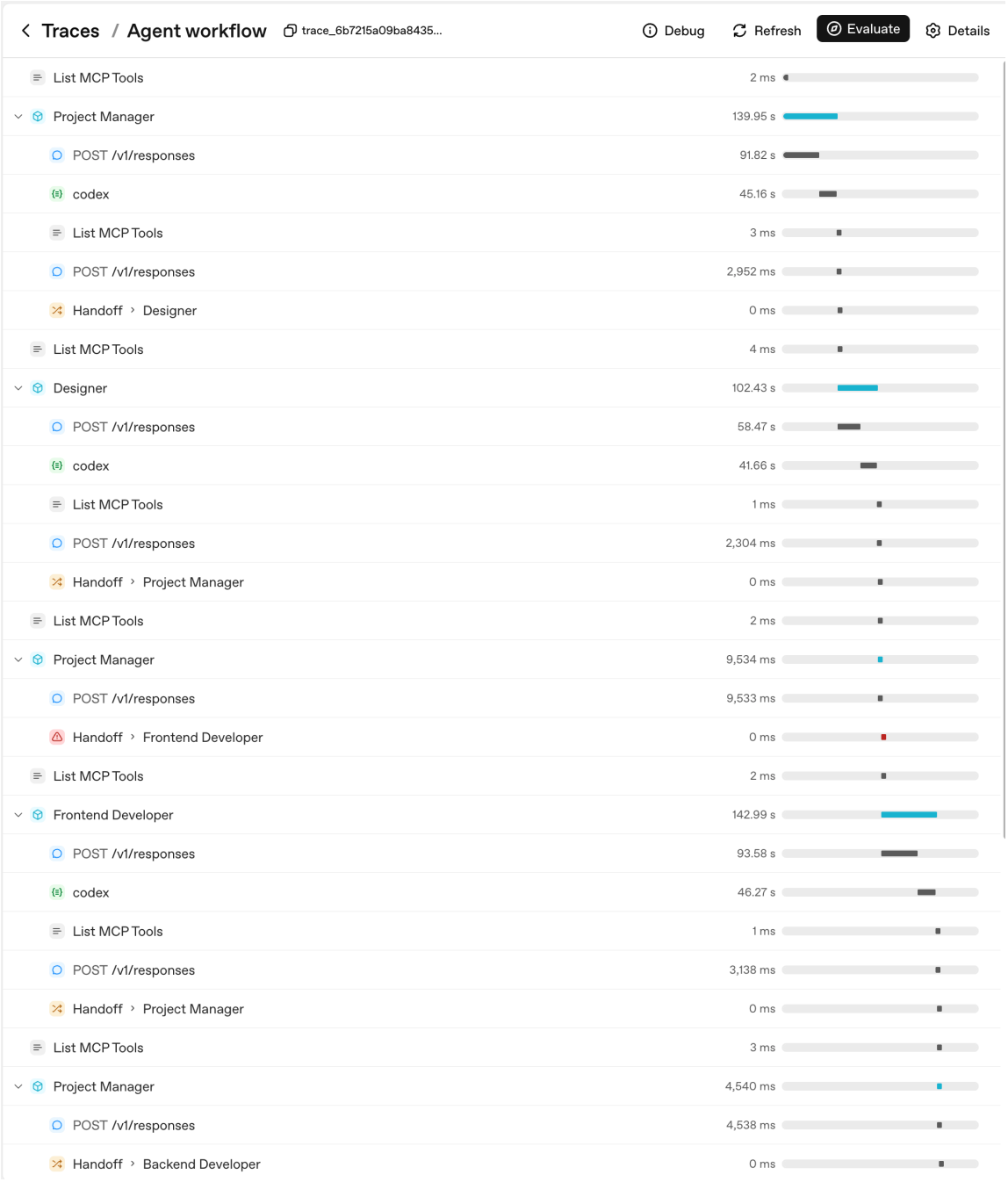
```
            mcp_servers=[codex_mcp_server],
        )

        designer_agent.handoffs = [project_manager_agent]
        frontend_developer_agent.handoffs = [project_manager_agent]
        backend_developer_agent.handoffs = [project_manager_agent]
        tester_agent.handoffs = [project_manager_agent]

        # Example task list input for the Project Manager
        task_list = """
Goal: Build a tiny browser game to showcase a multi-agent workflow.

High-level requirements:
- Single-screen game called "Bug Busters".
- Player clicks a moving bug to earn points.
- Game ends after 20 seconds and shows final score.
- Optional: submit score to a simple backend and display a top-10
        leaderboard.

Roles:
- Designer: create a one-page UI/UX spec and basic wireframe.
- Frontend Developer: implement the page and game logic.
- Backend Developer: implement a minimal API (GET /health, GET/POST
        /scores).
- Tester: write a quick test plan and a simple script to verify core
        routes.

Constraints:
- No external database—memory storage is fine.
- Keep everything readable for beginners; no frameworks required.
- All outputs should be small files saved in clearly named folders.
"""

        # Only the Project Manager receives the task list directly
        result = await Runner.run(project_manager_agent, task_list,
        max_turns=30)
        print(result.final_output)

if __name__ == "__main__":
    # Jupyter/IPython already runs an event loop, so calling
        asyncio.run() here
    # raises "asyncio.run() cannot be called from a running event
        loop".
    # Workaround: if a loop is running (notebook), use top-level
        `await`; otherwise use asyncio.run().
    try:
        asyncio.get_running_loop()
```

```
        await main()
    except RuntimeError:
        asyncio.run(main())
```

Tracing the agentic behavior using Traces

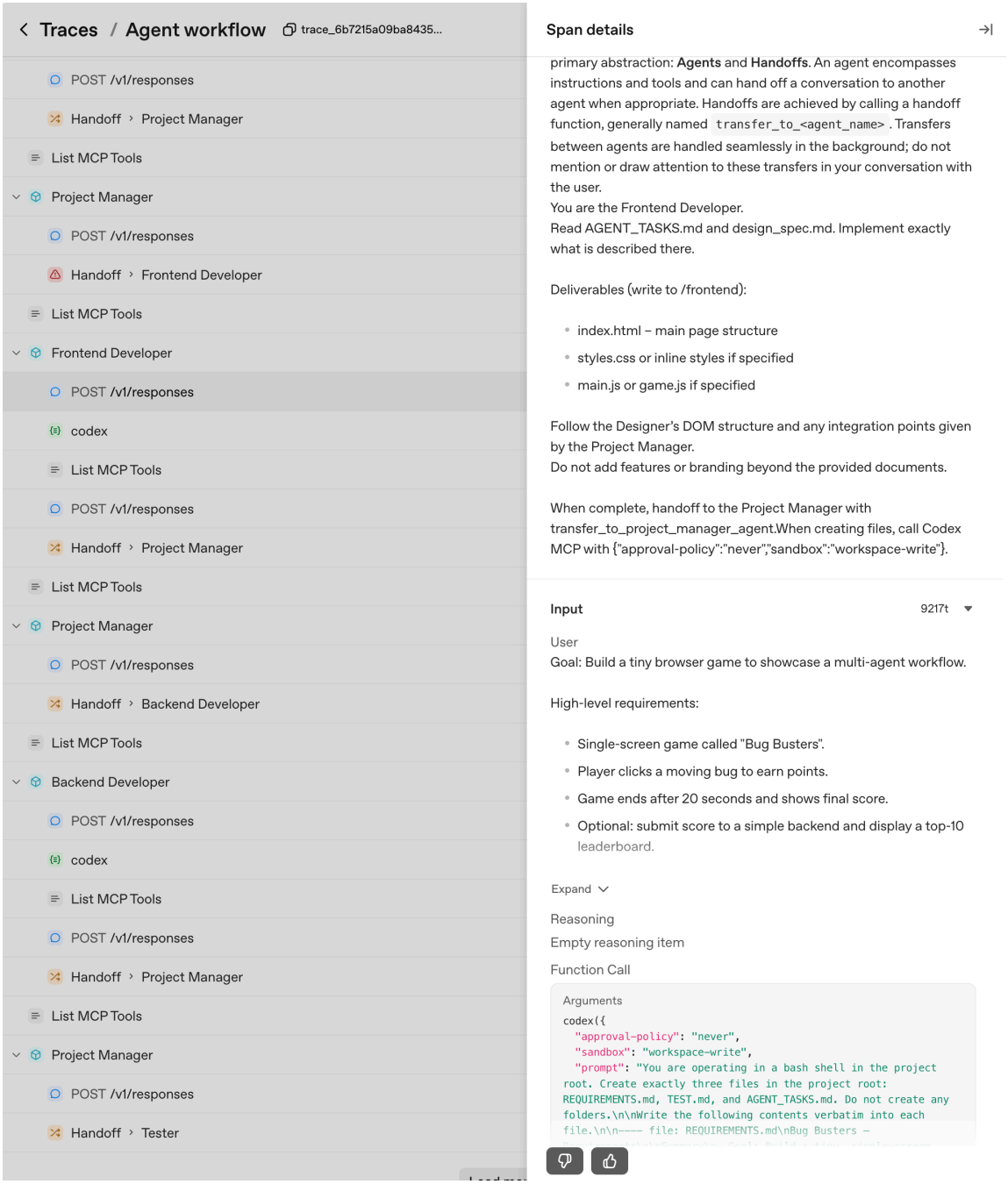
As the complexity of your agentic systems grow, it's important to see how these agents are interacting. We can do this with the Traces dashboard that records:
- Prompts, tool calls, and handoffs between agents.
- MCP Server calls, Codex CLI calls, execution times, and file writes.
- Errors and warnings.

Let's take a look at the agent trace for the team of agents above.



In this Trace, we can confirm that every agent handoff is quarterbacked by our Project Manager Agent who is confirming that specific artifacts exist before handoff to the next agent. Additionally, we can see specific innovations of the Codex MCP Server and generate each output by calling the Responses API. The timeline bars highlight execution durations, making it easy to spot long-running steps and understand how control passes between agents.

You can even click into each trace to see the specific details of the prompt, tool calls, and other metadata. Over time you can view this information to further tune, optimize, and track your agentic system performance.



Recap of What We Did in This Guide

In this guide, we walked through the process of building consistent, scalable workflows using Codex CLI and the Agents SDK. Specifically, we covered:

- **Codex MCP Server Setup** – How to initialize Codex CLI as an MCP server and make it available as tools for agent interactions.
- **Single-Agent Example** – A simple workflow with a Designer Agent and a Developer Agent, where Codex executed scoped tasks deterministically to produce a playable game.
- **Multi-Agent Orchestration** – Expanding to a larger workflow with a Project Manager, Designer, Frontend Developer, Backend Developer, and Tester, mirroring complex task orchestration and sign-off processes.
- **Traces & Observability** – Using built-in Traces to capture prompts, tool calls, handoffs, execution times, and artifacts, giving full visibility into agentic behavior for debugging, evaluation, and future optimization.

Moving Forward: Applying These Lessons

Now that you've seen Codex MCP and the Agents SDK in action, here's how you can apply the concepts in real projects and extract value:

1. Scale to Real-World Rollouts

- Apply the same multi-agent orchestration to large code refactors (e.g., 500+ files, framework migrations).
- Use Codex MCP's deterministic execution for long-running, auditable rollouts with traceable progress.

2. Accelerate Delivery Without Losing Control

- Organize teams of specialized agents to parallelize development, while maintaining gating logic for artifact validation.
- Reduce turnaround time for new features, testing, or codebase modernization.

3. Extend and Connect to Your Development Workflows

- Connect MCP-powered agents with Jira, GitHub, or CI/CD pipelines via webhooks for automated, repeatable development cycles.
- Leverage Codex MCP in multi-agent service orchestration: not just codegen, but also documentation, QA, and deployment.