ChatGPT

OpenAI Agents Go – Financial Research Example

The openai-agents-go SDK example `financial_research_agent` shows a multi-agent workflow in Go. The `FinancialResearchManager.Run` method orchestrates a sequence of sub-tasks: it calls a planner agent to generate a search plan, runs a search agent (in parallel) for each query term, then uses a writer agent (with specialist sub-agents as tools) to produce a report, and finally a verifier agent to check it [1]. Each step uses `agents.Run(ctx, Agent, input)` to invoke an agent and get its `FinalOutput` (e.g.

```
result, err := agents.Run(ctx, SearchAgent, inputData)
```

which returns structured output from the LLM [2]). The example also exposes "analysis" agents as callable tools: using `Agent.AsTool(...)` the code wraps a `FinancialsAgent` and `RiskAgent` as tools named `"fundamentals_analysis"` and `"risk_analysis"`, attaches these to the writer agent, and then runs the writer with those tools to generate the report [3] [4]. This demonstrates two-layer structure: a *manager* (orchestrator) driving the workflow, and multiple *agents* (including tools) handling sub-tasks.

Agent Patterns in the SDK

The Agents SDK is built around key concepts: **Agents** (LLMs with instructions and tools), **Handoffs** (transferring control between agents), and **Guardrails** (validation checks) [5]. The core agent loop works like this: call the LLM with the current agent's instructions, check the response – if it contains a final output (no tool calls or handoffs), return that; if it contains a *handoff*, switch to the new agent and continue; if it has *tool calls*, execute those tools and feed results back, then loop [6]. This makes it easy to implement complex flows: for example, the SDK's "handoffs" pattern lets one agent act as a triage or router. In the README's handoff example, a `triageAgent` inspects the query and uses `WithAgentHandoffs(spanishAgent, englishAgent)` to pass control to a Spanish or English specialist [7] [8]. Other patterns include parallelizing calls (as in the financial example's concurrent searches) and structuring outputs (the SDK supports setting an expected output type so it knows when to stop [6]). Overall, the `financial_research_agent` illustrates a **coordinated, multi-step agent workflow**: planning → parallel search → report writing with tools → verification [7] [1].

Comparing with Mosychlos Architecture

Mosychlos already uses a two-layer **orchestrator→engines** design. For example, the CLI commands create an Engine Orchestrator (`engine.New(...)`) and execute it as a pipeline [9]. The orchestrator builds a registry of `Engine` instances (e.g. a `RiskBatchEngine`) using dependency injection. Each engine encapsulates a multi-turn analysis: the `RiskBatchEngine`, for instance, repeatedly calls the LLM with custom prompts and processes its output via hooks into a shared state bag [10] [11]. In practice, an engine like Risk uses `base.BatchEngine` under the hood, which loops over "batch jobs" (e.g. analyzing different holdings) and collects results in the shared bag. This is conceptually similar to an agent loop, but Mosychlos separates concerns into distinct engine types (risk, allocation, etc.) that run one after another in the

pipeline. The openai-agents approach differs by packaging such flows into a single *agent* loop, but the goals overlap: both orchestrate LLM calls and tool usage. Importantly, as noted, Mosychlos aims to **keep the two-layer design**: the top-level orchestrator triggers engines, and each engine can itself invoke multiple agent-like operations. For example, the RiskBatchEngine's hooks gather data into the `sharedBag` [12] [11], just as an agent might write to a shared context. Thus, we can view each Mosychlos engine as analogous to one or more Agents SDK agents working on a task, all coordinated by the existing orchestrator pipeline [9] [1].

Leveraging Agent Handoffs ("HandsOff")

The OpenAI Agents Go SDK natively supports **agent handoffs**, a mechanism for one agent to transfer control to another [5]. In practical terms, you can configure an agent with `WithAgentHandoffs(...)` so that if the model decides a different specialist should take over, the framework switches to the target agent and continues the conversation [6]. The financial example hints at this (there is a "handoffs" example in the SDK), and in Mosychlos this could be powerful. For instance, one might create a top-level "triage" agent that examines the user's analysis request and then hands off to the appropriate domain agent (e.g. risk vs. market analysis). The SDK docs show exactly this pattern: the `Triage agent` example inspects a query and hands off to either a Spanish or English agent [8]. Applied to Mosychlos, a similar agent could route a portfolio query to a "Compliance agent" or a "Performance agent" based on intent. Because the SDK implements the handoff logic internally, it ensures context is passed seamlessly between agents. This "hands-off" feature is thus promising: it adds flexibility to the agent layer while still respecting the two-tier orchestrator/engine structure.

Integration Steps for Mosychlos

To integrate OpenAI's Agents SDK into Mosychlos, we recommend the following steps: 1. **Include the Agents SDK library.** Add `github.com/nlpodyssey/openai-agents-go` to the project (e.g. via `go get`). Ensure your Mosychlos configuration or environment provides any needed API keys (OpenAI key, etc.) to the Agents SDK.

2. **Define agent roles and tools.** For each analysis area (risk, allocation, compliance, etc.), create an `agents.Agent` with appropriate `WithInstructions(...)` and model settings. Wrap Mosychlos services as tools: for example, data-fetching functions or calculators can be registered as `agents.NewFunctionTool(...)` and attached via `WithTools(...)`. This mirrors how the financial example exposes analytics agents as tools [3].

3. **Modify engine implementations to use agents.** In the engine builder (e.g. in `internal/engine/wiring.go`), register engine factories that invoke these agents. For instance, a Risk engine's execution could call `result, err := agents.Run(ctx, myRiskAgent, inputData)` and then take `result.FinalOutput` as the analysis. You can see examples in the SDK: e.g., the `planSearches` method runs `agents.Run(ctx, PlannerAgent, "...")` and uses its output [1] [2]. In your engine's `Execute` or hooks, use `agents.Run` similarly. Update the shared bag or engine result with the agent's output.

4. **Maintain orchestrator control.** Keep the existing orchestrator pipeline intact: have it invoke the new "agent-based" engines in order. For example, the `Analyze` command already does `o := engine.New(...); o.Init(ctx); o.ExecutePipeline(ctx)` [9]. Under the hood, the Risk engine (now using an agent) will run as one node in this pipeline. This preserves the two-layer design: the orchestrator still sequences engines, each engine leverages agents.

5. **Optionally use agent handoffs.** If you need dynamic routing, implement a high-level "triage" or

“committee” agent. For example, inside an engine you could use `WithAgentHandoffs` so the agent can transfer to different sub-agents. Or insert an early engine that calls a triage agent and populates the bag to determine which branch to take. The handoff API simplifies passing context between agents (as shown in the SDK’s triage example [8]).

6. **Test and iterate.** Run Mosychlos workflows in interactive mode and check that agents produce the desired analysis. Use the SDK’s debugging features (like conversation tracing) to tune prompts. Ensure the agents’ outputs fit into your shared bag schema. Adjust tool usage, agent instructions, or maximum turns as needed. Because the Agents SDK supports specifying output types and max turns [6], you can control when an agent stops.

By following these steps, Mosychlos can leverage the OpenAI Agents Go framework under the hood. The orchestrator and engine layers remain, but each engine can internally use the multi-agent patterns from the SDK. In summary, we can integrate the SDK as a powerful engine runtime: calling `agents.Run` where we previously called `openai.ChatCompletion`, attaching tools for external data, and optionally chaining agents via handoffs. This fusion of Mosychlos’s orchestrator with the Agents SDK’s multi-agent workflows should enable richer, more flexible analysis while keeping the existing architecture intact [1] [5].

Sources: The examples and patterns are drawn from the OpenAI Agents Go SDK documentation and code [5] [6] [7] [1], and from the Mosychlos codebase (orchestrator and engine wiring) [9], as linked above. These illustrate how the agent loop, handoffs, and tool usage can be applied to the Mosychlos portfolio analysis domain.

---

[1] [2] [3] [4] manager.go
https://github.com/nlpodyssey/openai-agents-go/blob/a49f6d0880011341b6cd42f615e7838363b9fa9a/examples/financial_research_agent/manager.go

[5] [6] [7] [8] GitHub - nlpodyssey/openai-agents-go: A lightweight, powerful framework for multi-agent workflows in Go
https://github.com/nlpodyssey/openai-agents-go

[9] analyze.go
https://github.com/amaurybrisou/mosychlos/blob/e3a515260c38c39ea697329a438a232fec5a18de/cmd/mosychlos/analyze.go

[10] [11] [12] risk_batch.go
https://github.com/amaurybrisou/mosychlos/blob/e3a515260c38c39ea697329a438a232fec5a18de/internal/engine/risk/risk_batch.go