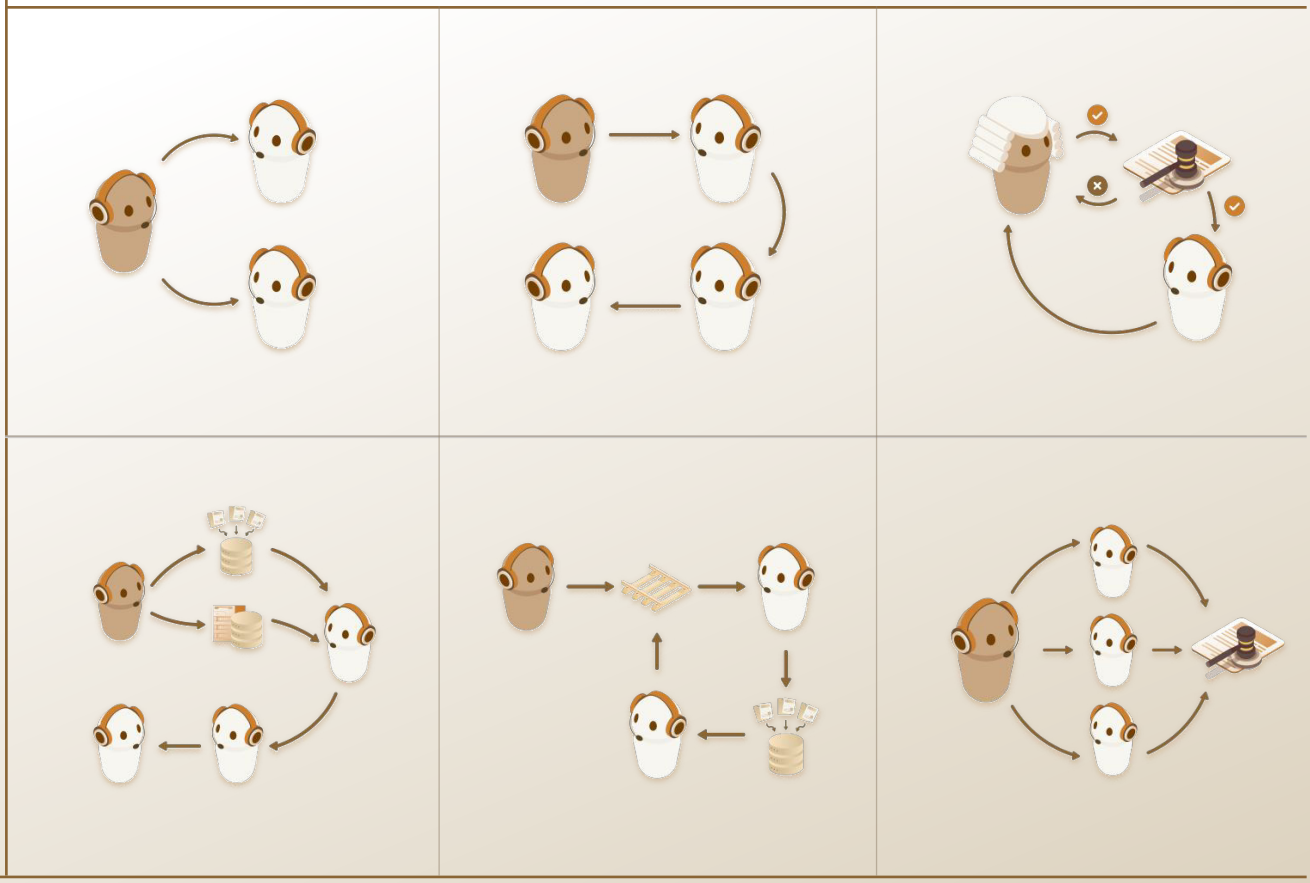
Agentic Patterns

6 Agentic AI Patterns

And 4 Enterprise AI Implementation Models



May 2025 Edition

Samuel Chan

Contents

Large Language Models (LLMs) to AI Agents

Large language models (LLMs) are a type of AI that can understand and generate human-like text. They are trained on vast amounts of data and can perform various tasks, such as answering questions, summarizing information, and generating creative content. LLMs are a key component of agentic AI systems, as they provide the natural language understanding and generation capabilities needed for these systems to interact with humans and the environment effectively.

AI Development companies like Supertype build LLM-powered applications that can be used in various industries, including:

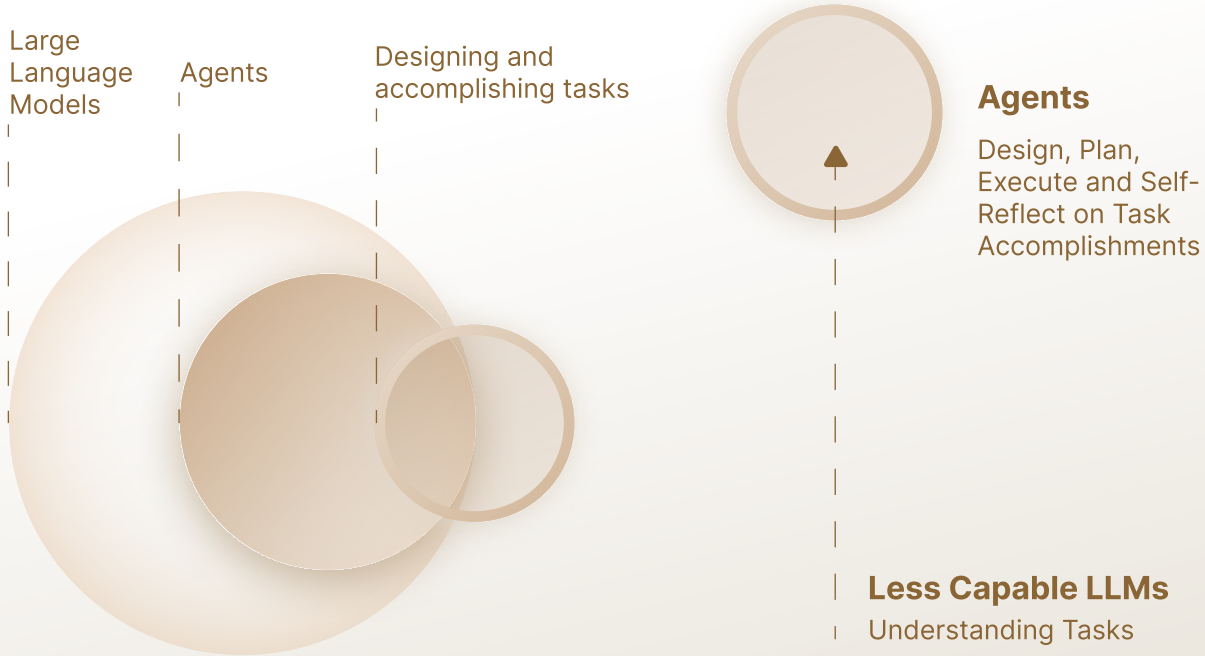
- Healthcare
- Finance
- Customer Service

These applications leverage the capabilities of LLMs to automate tasks, improve decision-making, enforce SOPs, and enable analytics-driven insights.

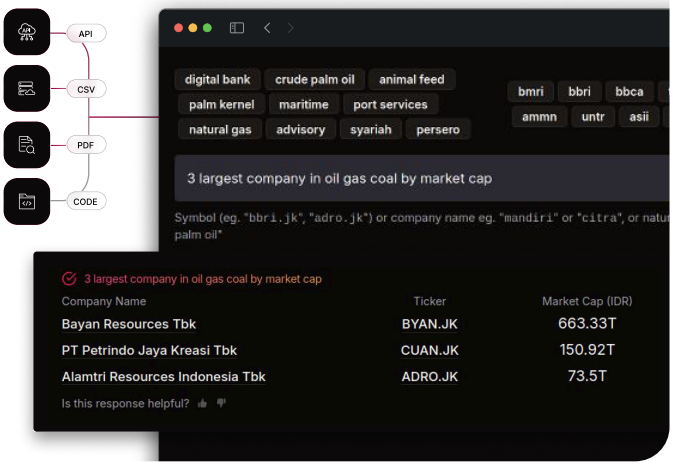
By leveraging LLMs, these applications are even capable of performing complex, sequential tasks that require interaction with multiple systems and tools. These action-taking capabilities are what make them agentic.

AI Agents

As alluded earlier, agentic AI is characterized by its **ability to take actions** in the world, rather than just providing information or generating text. This means that agentic AI systems can interact with their environment, make decisions, and perform tasks autonomously on behalf of the user. This is a marked departure from the conventional use of LLMs, which typically involve a human in the loop to interpret the output and take action. AI Agents, on the other hand, are designed to operate with a high degree of autonomy, from planning and decision-making to taking corrective actions.



Enterprise-Ready Agents



Enterprise-ready agents are AI systems that are specifically designed to meet the needs of businesses and organizations, and hence are built with a focus on:

Scalability	Enterprise-ready agents can handle large volumes of data and interactions, making them suitable for organizations of all sizes.
Security	These agents are built with robust security measures to protect sensitive data and ensure compliance with regulations.
Integration	Enterprise-ready agents can seamlessly integrate with existing systems and workflows, allowing organizations to leverage their existing infrastructure.
Robustness	These agents are designed to be reliable and resilient, ensuring that they can operate effectively in a variety of environments and conditions.
Customization	Enterprise-ready agents can be tailored to meet the specific needs and requirements of different organizations, allowing for greater flexibility and adaptability.

Building Blocks of Agentic AI Systems

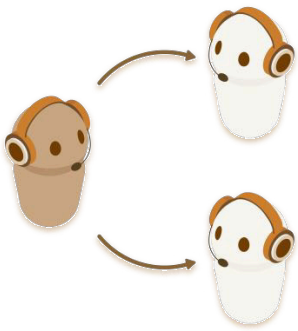
Agentic AI systems are built on a foundation of several key components, which work together to enable the system to take actions and make decisions autonomously. These components include:

Large Language Models (LLMs)	These models provide the natural language understanding and generation capabilities needed for the system to interact with humans and the environment effectively.
Knowledge Bases	These knowledge bases store the information and data needed for the agentic system to make informed decisions and take actions. They can include structured data, unstructured data, and domain-specific knowledge. Sectors Financial API is one such example of a LLM-ready knowledge base.
Actionable APIs	These APIs enable the agentic system to interact with external systems and services, allowing it to take actions and perform tasks autonomously. They can include APIs for data retrieval, data processing, and task execution.
Orchestration and Workflow Management	These components enable the agentic system to manage and coordinate the various tasks and actions it needs to perform, particularly when dealing with multiple agents across different task domains and enterprise systems.

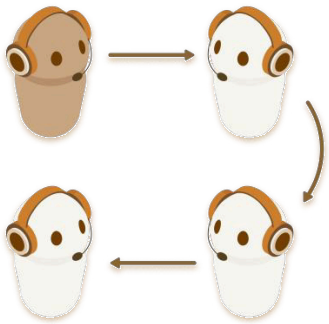
Over the next few sections, I will be demonstrating how these components come together to form agentic AI systems, and how the team at Supertype helps organizations develop and deploy these systems to meet their AI needs.

Agentic AI Patterns

Delegation & Hand Offs

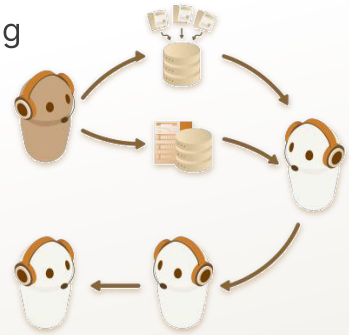


Sequential & Deterministic

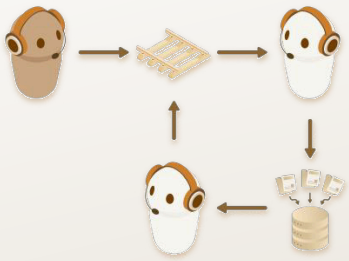


Judge & Critic

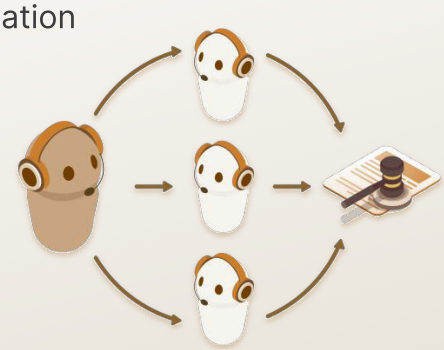
Tool-Use & Function Calling



Guardrails



Parallelization



Agent Foundations

Modular and Composability

At the core of agentic AI systems are small building blocks, each with its own set of **Model, Tools, Instructions and Guardrails** . At Supertype, we promote the use of modularity and composability in our agentic AI systems, making it easy for us and our clients to build out complex systems by combining or swapping out individual agents as needed to meet their specific AI needs.

These building blocks are designed to be modular and reusable, allowing developers to create complex systems by combining them in different ways. This modularity enables rapid development and deployment of agentic AI systems, as developers can easily swap out or modify individual components without having to redesign the entire system.

```python
from agents import Agent, Runner
from dotenv import load_dotenv

load_dotenv()

agent = Agent(name="Assistant",
              model="o3-mini",
              instructions="Answer as factually as possible.")

result = Runner.run_sync(agent, "Write a basic introduction about Supertype AI")
print(result.final_output)
```

In the code below, the Agent SDK will use the default model configured inopenai_provider.DEFAULT_MODEL (currently "gpt-4o") but is overwritten by ourmodel parameter in the Agent constructor.

LangChain · LlamaIndex · OpenAI Agent SDK

Code Examples on GitHub

I'm demonstrating the usage of these concepts using OpenAI's Agents SDK, but I also have code examples for LangChain and LlamaIndex for readers who prefer that.

https://github.com/onlyphantom/llm-python

Result of running the script above

Supertype AI is an innovative technology company specializing in artificial intelligence solutions designed to enhance business and operational efficiency. It offers a range of AI-powered tools and platforms that aim to streamline decision-making processes, automate repetitive tasks, and provide data-driven insights. By leveraging advanced machine learning algorithms, Supertype AI caters to various industries, helping organizations optimize their workflows and improve overall productivity.

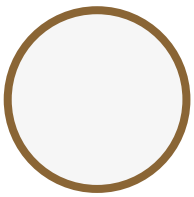
The official website for Supertype AI is [supertype.ai](https://www.supertype.ai).

The example above demonstrates a single-agent system, where we equip our agent with the right Instruction and Model (leaving out Tools and Guardrails – we will be covering both of these in later sections).

Input

"Write a basic introduction about Supertype AI"

Agent



Name:	Assistant
Model:	O3-Mini
Instructions:	"Answer as factually as possible."
Tools:	-
Guardrails:	-

Output

Supertype AI is an innovative technology company specializing in artificial intelligence solutions designed to enhance business and operational efficiency...The official website for Supertype AI is [supertype.ai] (https://www.supertype.ai).

Our orchestration uses the **Runner.run()** method to invoke a loop, which instructs the Agent to iterate through multiple cycles until an exit condition is triggered. An exit condition might be a tool-use (or function-calling), meeting the requirements of a structured output, hitting error, or reaching a max number of turns.

Tool Use and Function Calling for Finance LLMs
https://docs.sectors.app/recipes/generative-ai-python/02-tool-use

Structured Output with Tool Use LLMs
https://docs.sectors.app/recipes/generative-ai-python/03-structured-output

Starting with Single-Agent Systems

As far as complexity is concerned, the simplest AI system starts with a single agent, equipped with a single tool and a fixed prompt. The earlier example of the Assistant agent is a good example of this.

The next level of complexity is a single agent, but with multiple tools and the concept of prompt templates and tool selection. This is where the agent can choose which tool to use based on the input it receives. This pattern allows for greater flexibility and adaptability in the system's behavior without requiring us to scale up the number of agents.

```
customer_service_manager = Agent(
    name="Financial Analyst Support",
    instructions="You are a financial analyst at a stock brokerage firm. You are interacting
with {{user_first_name}} who speak {{user_preferred_language}} and whose account number is
{{user_account_number}}. You are to assist {{user_first_name}} with their account-related
queries, in {{user_preferred_language}}.",
    tools=[...]
)
```

Scaling up to Multi-Agent Systems

A multi-agent system might have deployed multiple agents, with each agent instructed to answer a specific question, or strictly answer in a specific language.

In the example above however, the single-agent system is being fed with a prompt template that is dynamically populated with user-specific information. This allows the agent to provide personalized responses based on the user's account number and preferred language, without necessarily having to deploy a multi-agent (i.e. one for each language) system.

Supertype's recommendation is to start with a single agent, and only scale up to a multi-agent system when the complexity of the task requires it. An obvious example of this is when the tasks that your agents are performing are completely independent of each other, and can be run in parallel -- this is the Parallelization Pattern, which we will cover in a later section.

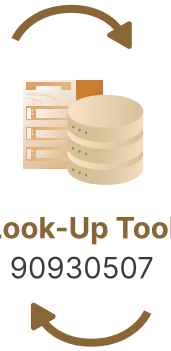
"What is my account balance?"

Context

Name: Samuel Chan
Customer Number: 90930507

Manager

Name: Manager
Model: O3-Mini
Instructions:
"Retrieve the necessary information and delegate accordingly"
Tools: DB Look-Up Tool
Guardrails: NO-MANUAL-ENTRY-OF-CUSTOMER-NUMBER



Look-Up Tool
90930507

Context

Name: Samuel Chan
Customer Number: 90930507

Savings Account
90930507-SAV-01

Fixed Deposits Account
42309-32-DEP-00-77

Sub-Agent 1

Name: Savings Account Clerk
Model: O3-Mini
Instructions: "Read account balance of {{0930507-SAV-01}}"
Tools: DB Look-Up Tool
Guardrails: NO-UPDATE-DELETE-OPS

Sub-Agent 2

Name: Deposits Account Clerk
Model: O3-Mini
Instructions: "Sum all outstanding balances in {{42309-32-DEP-00-77}}"
Tools: DB Look-Up Tool, Sum Calculator
Guardrails: NO-UPDATE-DELETE-OPS

Manager

Name: Manager
Model: gpt-4o
Instructions: "Use calculator to sum. Reply to user factually."
Tools: Sum Calculator Tool
Guardrails: NO-LEAK-SENSITIVE-INFO

Context

Name: Samuel Chan
Customer Number: 90930507

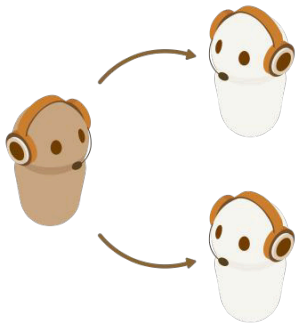
Savings Account
90930507-SAV-01

Fixed Deposits Account
42309-32-DEP-00-77

Balance
Savings: S$4,501.93
Deposits: S$93,000.00

"You have $4,501.93 in your Savings Account and $93,000 in Fixed Deposits"

The Hand-off and Delegation Pattern



Decentralized Hand-Offs Pattern

Mirroring the diagram above, we can implement the Hand-Off and Delegation Pattern by creating multiple sub-agents that are responsible for different tasks or domains. In the example below, the **customer_service_manager** agent can choose to delegate to one of the two sub-agents, depending on the user's preferred language. A well-designed hand-off and delegation pattern allow for separation of concerns and modularity, while keeping each agent loosely coupled to the main agent.

```python
# import os
from agents import Agent, Runner
from dotenv import load_dotenv

load_dotenv()

german_agent = Agent(
    name="German Assistant",
    instructions="Always respond in German. Be polite and concise.",
)

english_agent = Agent(
    name="English Assistant",
    instructions="Always respond in English.",
)

customer_service_manager = Agent(
    name="Customer Service Manager",
    instructions="Handoff to the appropriate agent based on the language of the request.",
    handoffs=[german_agent, english_agent],
)

async def main():
    query = "Hallo, ich habe ein Problem und muss mit dem Manager sprechen"
    result = await Runner.run(
        customer_service_manager,
        query
    )
    print(f"👧: {query}")
    print(f"🤖: {result.final_output}")

if __name__ == "__main__":
    import asyncio
    asyncio.run(main())
```

Hand-Off vs the Manager Pattern

The Hand-Off Pattern is similar to the Manager Pattern, but with a key difference. In the Manager Pattern, the manager agent is responsible for coordinating the actions of multiple sub-agents, and typically consolidates the results from these sub-agents to further synthesize the final output.

In the Hand-Off pattern as implemented by the Agent SDK however, the agents are symbolically peers of each other instead of having a manager-subordinate relationship. An agent handing off a task to another will fully transfer the conversation state to the delegated agent, and the execution flow immediately switches to the delegated agent.

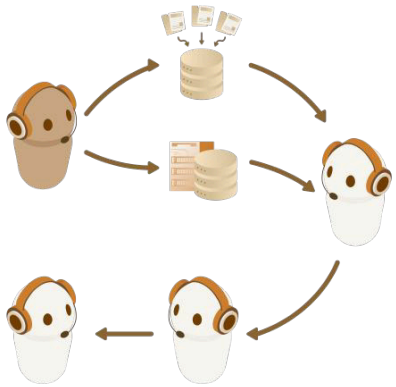
The Hand-Off pattern is useful when agents are specialized to take on the full control of specific tasks **without needing a central coordinator to be involved.** If necessary, we can always implement another hand-off to pass the baton to another agent, or even to the original agent.

Result of running the script above

👧: Hallo, ich habe ein Problem und muss mit dem Manager sprechen

🤖: Natürlich, ich kann Ihnen helfen. Was genau ist Ihr Anliegen? Dann kann ich es an den Manager weiterleiten.

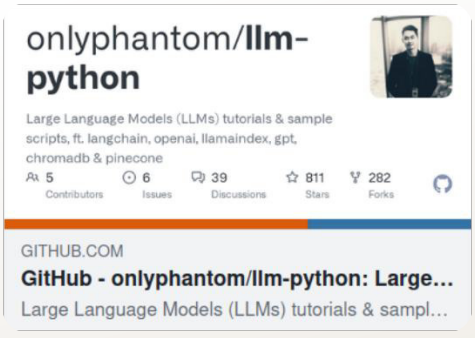
The Tool-Use Pattern



Tool-Use and Function Calling

Tool-Use and Function Calling is one of the most powerful patterns in agentic AI systems, as it allows agents to gain an understanding of the world beyond what their training data provides. This pattern also unlock the agent's ability to take actions in the world, such as making API calls to retrieve data, perform calculations, surf the web, write an investment thesis to a markdown file, or even send emails.

The Agent SDK provides a **function_tool** decorator that allows us to define functions that can be called by the agent. This is similar to LangChain, LLamaIndex and other libraries that provide similar functionality, so feel free to use the code reference from my LLM-Python course (free, and fully open-sourced) if you are already familiar with it.



https://github.com/onlyphantom/llm-python

https://docs.sectors.app/recipes/generative-ai-python/02-tool-use

Preparing our Agents for Tool-Use

Below is some scaffolding code to get you started with the full code example of the Tool-Use and Function Calling Pattern. It uses a secret key (typically stored in a .env file) to authenticate with Sectors Financial API, a LLM-ready knowledge base that Supertype has built to help organizations build financial-aware AI systems. Through the Sectors API, we can build AI agents capable of performing complex financial tasks, such as portfolio management, risk assessment, and investment research.

The usage of this external API is a good example of the Tool-Use and Function Calling Pattern, where our agent relies on an external tool to be "intelligent" and take informed actions on behalf of the user.

```python
import os
import json
import requests

from dotenv import load_dotenv

load_dotenv()
SECTORS_API_KEY = os.getenv("SECTORS_API_KEY")

headers = {"Authorization": SECTORS_API_KEY}
def retrieve_from_endpoint(url: str) -> dict:
    try:
        response = requests.get(url, headers=headers)
        response.raise_for_status()
        data = response.json()
    except requests.exceptions.HTTPError as err:
        raise SystemExit(err)
    return json.dumps(data)


async def main():
    pass

if __name__ == "__main__":
    import asyncio
    asyncio.run(main())
```

Equipping our Company Research Agent with Tools

We equip our single agent with two tools, **get_company_overview** and **get_top_companies_ranked**, allowing it to pick the best tool to use based on the user's query. The agent will then call the appropriate function to retrieve the data from Sectors API, and return the result to the user.

```python
from agents import Agent, Runner, function_tool

@function_tool
def get_company_overview(ticker: str, country: str) -> str:
    """
    Get company overview from Singapore Exchange (SGX) or Indonesia Exchange (IDX)
    """
    assert country.lower() in ["indonesia", "singapore", "malaysia"],
      "Country must be either Indonesia, Singapore, or Malaysia"

    if(country.lower() == "indonesia"):
        url = f"https://api.sectors.app/v1/company/report/{ticker}/?sections=overview"
    if(country.lower() == "singapore"):
        url = f"https://api.sectors.app/v1/sgx/company/report/{ticker}/"
    if(country.lower() == "malaysia"):
        url = f"https://api.sectors.app/v1/klse/company/report/{ticker}/"

    try:
        return retrieve_from_endpoint(url)
    except Exception as e:
        print(f"Error occurred: {e}")
        return None


@function_tool
def get_top_companies_ranked(dimension: str) -> List[str]:
    """
    Return a list of top companies (symbol) based on certain dimension
        (dividend yield, total dividend, revenue, earnings, market cap,
        PB ratio, PE ratio, or PS ratio)
    """

    url = f"https://api.sectors.app/v1/companies/top/?classifications={dimension}"

    return retrieve_from_endpoint(url)


company_research_agent = Agent(
    name="company_research_agent",
    instructions="Research the company based on the ticker provided.",
    tools=[get_company_overview, get_top_companies_ranked],
    tool_use_behavior="run_llm_again"
)

async def main():
    query = "Tell me about the company listed on Singapore Exchange with ticker 'D05'."
    result = await Runner.run(
        company_research_agent,
        query
    )
    print(f"👧: {query}")
    print(f"🤖: {result.final_output}")

if __name__ == "__main__":
    import asyncio
    asyncio.run(main())
```

The workflow of our Agent can be broadly described as the following:

- Decide which tool to use based on the intent of the user's query
- Call the tool with the appropriate parameters
- When the tool returns a result, run the LLM on the result to process it into a human-readable format (**run_llm_again** as opposed to **stop_on_first_tool** in **tool_use_behavior**), then return the final output to the user.

Here is a result of running the script. Instead of making up an answer through linguistic mimicry or past training data, our Agent activates the right tool and format the API response into a beautiful, Markdown-formatted answer:

```
samuel@pc:~/work/workshops/ai-search/may$ python 2_tooluse.py
👧: Tell me about the company listed on Singapore Exchange with ticker 'D05'.
🤖: DBS Group Holdings Ltd (Symbol: D05) is a major financial services group based in
Singapore, operating in the "Banks - Regional" subsector.

### Overview
- **Market Cap**: SGD 119.66 billion
- **Recent Volume**: 5,418,900 shares
- **Employees**: 41,000

### Financial Performance
- **Earnings (TTM)**: SGD 11.27 billion
- **Revenue (TTM)**: SGD 22.3 billion
- **Price to Earnings (P/E)**: 10.68
- **Price to Sales (P/S)**: 5.52
- **Price to Book (P/B)**: 1.8

### Stock Movement
- **Change(1 Day)**: -0.52%
- **Change(1 Year)**: +28.17%
- **52 Week High**: SGD 46.02 (Feb 26, 2025)
- **52 Week Low**: SGD 31.39 (Aug 6, 2024)

### Dividends
- **Dividend Yield (5 Year Average)**: 4.59%
- **Dividend TTM**: SGD 2.22
- **Dividend Growth Rate**: -71.58%

### Historical Earnings
- **2023**: SGD 10.06 billion
- **2024**: SGD 11.29 billion

### Dividend History
- **2024 Total Dividend**: SGD 2.11 per share
- **2025 Partial**: SGD 0.6 per share as of April 7, 2025

DBS has shown robust growth over multiple time frames, with notable long-term improvements in
both earnings and revenue. The company also provides consistent dividends to its
shareholders.
```

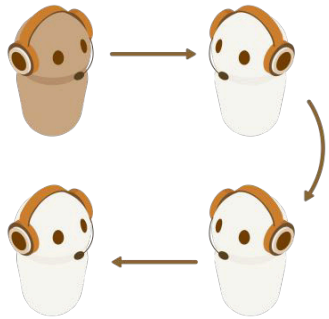
Tool-Use Behavior by Agents

Notice that the tool is never called directly by the user. The agent is responsible for picking the right tool to use, calling it with the right parameters, and processing the result into a human-readable format -- all with little to no human intervention.

When a tool requires multiple parameters, the agent (powered by a language model capable of understanding the user's intent) will make its best guess on what these parameters should be, and call the tool with those parameters. In the example above, it needs to figure out the ticker symbol of the company and the country where said company is listed to call the **get_company_overview** tool.

The raw output of the tool (see image on this page) is not easily readable by humans, so the Agent calls the LLM again (**run_llm_again**) to turn this into a report that is more suitable for human consumption.

The Deterministic and Sequential Chain Pattern



LangChain got its start with the idea of chaining together LLMs and tools to create complex workflows. The Deterministic and Sequential Chain Pattern is a demonstration of this idea, where we can create a chain of agents that work together to achieve a common goal. The output of one agent is passed as input to the next agent in the chain, allowing for a seamless flow of information and actions that unlock powerful combinations for enterprise productivity.

When designing sequential chains, Supertype recommends that the developer pays **special attention to the overall orchestration and respective outputs of each agent** in the chain.

The order in which the agents are executed can have a significant impact on the overall performance and effectiveness of the system, as the output of one agent is typically passed as input to the next; An incorrect or inaccurate output can hence lead to cascading errors throughout the chain.

In this orchestration, we have set up three tools:

- **get_top_companies_ranked**
- **get_company_overview**
- **csv_export tool**

We have also set up three Agents, two of which are equipped with tools to make external API calls to fetch live financial data.

```python
from pydantic import BaseModel
from typing import List
from agents import Agent, Runner, function_tool, trace
from utils.api_client import retrieve_from_endpoint
from datetime import datetime


@function_tool
def get_company_overview(ticker: str, country: str) -> str:
    """
    Get company overview from Singapore Exchange (SGX) or Indonesia Exchange (IDX)
    """
    assert country.lower() in ["indonesia", "singapore", "malaysia"], "Country must be either
Indonesia, Singapore, or Malaysia"

    if(country.lower() == "indonesia"):
        url = f"https://api.sectors.app/v1/company/report/{ticker}/?sections=overview"
    if(country.lower() == "singapore"):
        url = f"https://api.sectors.app/v1/sgx/company/report/{ticker}/"
    if(country.lower() == "malaysia"):
        url = f"https://api.sectors.app/v1/klse/company/report/{ticker}/"

    try:
        return retrieve_from_endpoint(url)
    except Exception as e:
        print(f"Error occurred: {e}")
        return None


@function_tool
def get_top_companies_ranked(dimension: str) -> List[str]:
    """
    Return a list of top companies (symbol) based on certain dimension
        (dividend yield, total dividend, revenue, earnings, market cap,
        PB ratio, PE ratio, or PS ratio)
    """

    url = f"https://api.sectors.app/v1/companies/top/?classifications={dimension}&n_stock=3"

    return retrieve_from_endpoint(url)

@function_tool
def csv_export(data: object) -> str:
    """
    Convert the object to a csv
    """
    import pandas as pd

    # Convert the object to a DataFrame
    df = pd.DataFrame.from_dict(data, orient='index')

    # Convert the DataFrame to a CSV saved as 'export.csv'
    df.to_csv('export.csv', index=True)
    return "Successfully exported to export.csv"

class ValidTickers(BaseModel):
    tickers: List[str]


get_top_companies_based_on_metric = Agent(
    name="get_top_companies_based_on_metric",
    instructions="Get the top companies based on the given metric. Return the tickers of the top
companies, without the .JK suffix. Return in a List.",
    tools=[get_top_companies_ranked],
    output_type=List[str],
)

determine_companies_to_research = Agent(
    name="determine_companies_to_research",
    instructions="Generate a list of tickers (symbols) of companies to research based on the query.
Tickers on IDX are exactly 4 characters long, e.g. BBCA, BBRI, TKLM",
    output_type=ValidTickers,
)

company_research_agent = Agent(
    name="company_research_agent",
    instructions="Research each company of a given list using the assigned tool, always assume
indonesian companies unless otherwise specified.",
    tools=[get_company_overview],
    output_type=str
)
```

We expect our entry point to be the **determine_companies_to_research** agent, which will call the **get_top_companies_ranked** tool to retrieve a list of companies based on our screening criteria.

The output of this tool is vetted to make sure it is of the right format (list of tickers), and if it is, the agent will call the **get_company_overview** tool repeatedly for each of the tickers, cleverly substituting the ticker symbol into the tool call. The output of this second agent can be further processed by the LLM to generate a human-readable report, or be exported to a CSV file through the **csv_export** tool.

```
async def main():
    input_prompt = input(f"🤖: What kind of companies are you interested in? \n👧: ")
    # Ensure the entire workflow is a single trace
    with trace("Deterministic research flow"):
        # 1. Determine the companies ranked by certain dimension
        top_companies_ranked = await Runner.run(
            get_top_companies_based_on_metric,
            input_prompt
        )
        print("🤖:", top_companies_ranked.final_output)


        # 2. Add a gate to stop if the tickers are not valid
        assert isinstance(
            top_companies_ranked.final_output, list), "Invalid tickers"

        # 3. Research the company based on the query
        # 3.1 Append to Notes
        with open("3_research_notes.txt", "a") as f:
            f.write(f"Research on {input_prompt}:\n")
            f.write(f"Top companies: {top_companies_ranked.final_output}\n\n\n")

            for ticker in top_companies_ranked.final_output:
                print(f"🤖: Getting information on: {ticker}")
                company_research_result = await Runner.run(
                    company_research_agent,
                    ticker
                )
                if not company_research_result or not company_research_result.final_output:
                    print(f"🤖: Failed to get data for {ticker}")
                    f.write(f"❌ Failed to get data for {ticker}\n")
                    continue
                print(f"🤖: {company_research_result.final_output}")
                f.write(f"Company: {ticker}\n")
                f.write("" + company_research_result.final_output + "\n\n" + "Research Date: " +
datetime.now().isoformat() + "\n\n\n")


        print(f"🤖: Done! I have provided the information on: {input_prompt}")
```

Stock Screening to Report in Seconds

Depending on whether the downstream agent is asked to write to a CSV file or generate a human-readable text report, you should still be able to see the multi-agent orchestration in action. The first agent determines a list of companies to research through its own tool-use, and the output of this gaent is used as an input to a second agent, which in turn calls its own tool to retrieve stock reports from Sectors Financial API.

The benefit of this pattern is that we are able to support complex, multi-step workflows that would otherwise be time-consuming for a human to perform manually. Instead, we are able to fully automate the process, and get an insightful report (or CSV) in a matter of seconds by having multiple agents working together in a coordinated manner.

As we will see later, we can further improve the performance of this pattern by incorporating parallelization, allowing multiple agents to retrieve financial data simultaneously and write to the CSV or text report in parallel. This is particularly useful when dealing with large volumes of data or when the agents need to perform time-consuming tasks, such as web scraping or data processing.

```
🏦: What kind of companies are you interested in?
👱: companies with highest market cap
🏦: ['BBCA', 'BREN', 'TPIA']
🏦: Passed Assertion
🏦: Getting information on: BBCA
🏦: **PT Bank Central Asia Tbk. (BBCA.JK)**

- **Industry:** Financials
- **Sub-Industry:** Banks
- **Sector:** Banks
- **Market Cap:** IDR 1,075,565,641,596,928
- **Market Cap Rank:** 1
- **Employees:** 24,685

**Listing Details**

- **Board:** Main
- **Listing Date:** 31 May 2000

**Contact Information**

- **Address:** Menara BCA, Grand Indonesia, Jalan MH Thamrin
No. 1, Jakarta 10310
- **Phone:** 021-23588000
- **Email:** investor_relations@bca.co.id
- **Website:** [www.bca.co.id](http://www.bca.co.id)

**Stock Information**

- **Last Close Price:** IDR 8,725 (as of 29 April 2025)
- **Daily Change:** -0.57%

**Price History**

- **YTD Low:** IDR 7,275 (8 April 2025)
- **YTD High:** IDR 9,925 (3 January 2025)
- **52-Week High:** IDR 10,950 (23 September 2024)
- **All-Time High:** IDR 10,950 (23 September 2024)
- **All-Time Low:** IDR 175 (8 June 2004)

**ESG Score:** 21.5

Feel free to ask if you need more information!
🏦: Getting information on: BREN
🏦: ### PT Barito Renewables Energy Tbk. (BREN.JK)

**Overview:**
- **Listing Board:** Main
- **Industry:** Electric Utilities
- **Sub-Industry:** Electric Utilities
- **Sector:** Infrastructures
- **Sub-Sector:** Utilities
- **Market Cap:** IDR 806,060,671,631,360
- **Market Cap Rank:** 2

**Contact Information:**
- **Address:** Wisma Barito Pacific II, Lantai 23, Jl. Let.
Jend. S. Parman Kav. 60, RT 010, RW 005, Slipi, Palmerah,
Jakarta 11410, Indonesia
- **Phone:** (021) 530 6711
- **Email:** corpsec@baritorenewables.co.id
- **Website:** [www.baritorenewables.co.id](http://
www.baritorenewables.co.id)

**Employment:**
- **Number of Employees:** 638

**Financial Highlights:**
- **Listing Date:** 2023-10-09
- **Last Close Price:** IDR 6,025 (as of 2025-04-29)
- **Daily Close Change:** -0.41%

**Price Information:**
- **All Time Low:** IDR 975 (2023-10-09)
- **All Time High:** IDR 12,200 (2024-05-17)
- **Year-to-Date Low:** IDR 4,170 (2025-04-09)
- **Year-to-Date High:** IDR 10,650 (2025-01-08)

**ESG Score:**
- **ESG Score:** 37.7

If you need more information, feel free to ask!
🏦: Getting information on: TPIA
🏦: ### PT Chandra Asri Pacific Tbk (TPIA.JK)

- **Industry**: Chemicals
- **Sub-Industry**: Basic Chemicals
- **Sector**: Basic Materials
- **Market Cap**: IDR 663,975,771,504,640
- **Market Cap Rank**: 3
... [truncated]

#### Contact Information
- **Address**: Gedung Wisma Barito Pacific Tower A, Lt. 7 Jl.
Let Jend S. Parman Kav. 62-63, RT.008 RW.004, Slipi Palmerah,
Jakarta Barat, DKI Jakarta - 11410
- **Phone**: (021) 5307950
- **Email**: corporatesecretary@capcx.com
- **Website**: [www.chandra-asri.com](http://www.chandra-
asri.com)
🏦: Done! I have provided the information on: top market cap
```

The Judge and Critic Pattern

In this illustration of the Judge and Critic pattern, the first agent generates a stock summary from the research notes and the second agent evaluates the summary. The first agent is asked to continually improve the summary until the evaluator gives a pass.

3_research_notes.txt is the text file generated by our previous section where our multi-agent orchestration pattern is demonstrated. As is expected, you will need a tool to read from the stock research text file, and an Agent capable of using it.

```python
@function_tool
def read_company_data_from_txt() -> str:
    """
    Read company data from the text file 3_research_notes.txt
    """
    try:
        with open("3_research_notes.txt", "r") as file:
            data = file.read()
            print(data)
            return data
    except FileNotFoundError:
        return "File not found. Please ensure the file exists."
    except Exception as e:
        return str(e)

read_company_data_from_txt = Agent(
    name="read_company_data_from_txt",
    instructions=(
        "Given a company name or ticker by the user, read the company data from the text file
3_research_notes.txt"
        "Summarize them into 2-3 paragraphs and be informative so it reads like a professional
report."
        "If there is any feedback, incorporate them to improve the report. If the ticker is not
found, say so."
    ),
    tools=[read_company_data_from_txt],
)

@dataclass
class EvaluationFeedback:
    feedback: str
    score: Literal["pass", "expect_improvement", "fail"]
```

With an Agent responsible for writing a summary report, we then implement another Agent and call it **evaluator**. This agent is responsible for evaluating the output of the first agent, and providing (1) feedback on the quality of the report and (2) score it between **pass, expect_improvement**, and **fail**. This is a complete example of the Judge and Critic Pattern, and one that loosely borrow from established concepts in reinforcement learning.

```
evaluator = Agent[None](
    name="evaluator",
    instructions=(
        "You evaluate a stock overview summary and decide if it's good enough."
        "If it's not good enough, you provide feedback on what needs to be improved."
    ),
    output_type=EvaluationFeedback,
)

async def main() -> None:
    msg = input("🤖: What company are you interested in? \n👧: ")
    input_items: list[TResponseInputItem] = [{"content": msg, "role": "user"}]

    summary: str | None = None

    with trace("LLM as a judge"):
        while True:
            summarized_results = await Runner.run(
                read_company_data_from_txt,
                input_items,
            )

            input_items = summarized_results.to_input_list()
            summary = ItemHelpers.text_message_outputs(summarized_results.new_items)
            print("Stock overview summary generated")

            evaluator_result = await Runner.run(evaluator, input_items)
            result: EvaluationFeedback = evaluator_result.final_output

            print(f"Evaluator score: {result.score}")

            if result.score == "pass":
                print("The stock summary is 💡!, exiting.")
                break

            print("Re-running with feedback")

            input_items.append({"content": f"Feedback: {result.feedback}", "role": "user"})

    print(f"Final Summary: {summary}")
    print("Input items:", input_items)
```

Each time the **evaluator** agent is called, it will evaluate the output of the report-writing agent and provide feedback on the quality of the report.

{'content': 'Feedback: The overview of PT Bank Central Asia Tbk (BBCA) provides comprehensive details about its market position, stock performance, and ESG score. However, to enhance clarity and usefulness, it could benefit from the following improvements:\n\n1. **Comparative Analysis**: Include a short comparison with other top companies in the banking sector to provide context about BBCA's performance.\n \n2. **Growth Prospects**: Discuss any recent strategic initiatives or partnerships that might influence future growth or stability.\n\n3. **Financial Highlights**: Briefly state key financial indicators like revenue or net profit from recent quarters to add more depth.\n\n4. **Visual Elements**: Integrate charts or graphs for stock price trends over time to visually emphasize points.\n\nIncorporating these elements would create a richer, more insightful summary for potential investors or stakeholders.', 'role': 'user'},

{'content': 'Feedback: The summary of PT Bank Central Asia Tbk (BBCA) provides valuable insights into its market position and strategic initiatives, offering depth and context. To enhance the overview further, consider these improvements:\n\n1. **Risk Factors**: Include potential risks or challenges BBCA might face, such as regulatory changes or economic downturns, to offer a balanced view.\n \n2. **Customer Demographics**: Provide insights into key customer segments or demographics that BBCA primarily serves, highlighting its market strategy.\n\n3. **Competitive Landscape**: Further expand on competitors' analysis, detailing specific areas where BBCA holds advantages or faces competition.\n\n4. **Digital Transformation**: Dive deeper into their digital transformation initiatives and achievements for a comprehensive look at their tech-forward strategies.\n\nAdding these aspects will create a rounded, well-informed summary, offering stakeholders a clearer picture of opportunities and risks associated with BBCA.', 'role': 'user'},

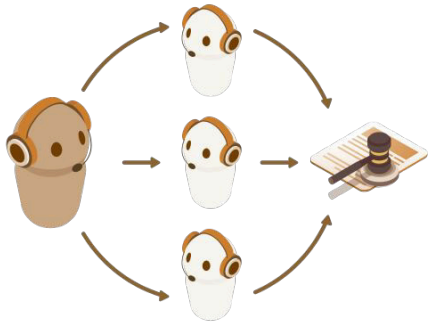
{'content': "Feedback: The summary of PT Bank Central Asia Tbk (BBCA) is comprehensive and informative, covering critical areas such as market position, strategic growth, and digital transformation efforts. However, to refine the overview, additional emphasis on BBCA's adaptability in a volatile market could be beneficial. Further enhancement might include:\n\n1. **Case Studies or Examples**: Cite specific examples or case studies of successful digital initiatives to provide concrete evidence of BBCA's evolution.\n \n2. **Analyst Predictions**: Integrate predictions or insights from market analysts regarding BBCA's future trajectory, offering expert perspectives.\n\n3. **Long-term Trends**: Discuss long-term economic or technological trends that could impact BBCA, anchoring its current initiatives within a broader context.\n\nIncluding these could further enrich the narrative and provide a balanced, forward-looking perspective for potential investors or stakeholders.", 'role': 'user'}

The report-writing agent will then use this feedback to refine its output, rewriting the report to improve its quality. In larger systems, the pattern can be extended to include multiple evaluators, each with its own set of criteria, fact-checking and feedback mechanisms.

This allows for a more comprehensive evaluation of the report and helps to ensure that the final output meets the desired quality standards with various levels of fact-checking and review processes.

```
samuel@pc:~/work/workshops/ai-search/may$ python 4_judge_critic.py
🤖: What company are you interested in?
👴: bbca
Company: BBCA
**PT Bank Central Asia Tbk. (BBCA.JK)**
- **Industry:** Financials
- **Sub-Industry:** Banks
- **Sector:** Banks
- **Market Cap:** IDR 1,075,565,641,596,928
- **Market Cap Rank:** 1
- **Employees:** 24,685
...
2025-04-30T13:41:03.998111
Stock overview summary generated
Evaluator score: expect_improvement
Re-running with feedback
Stock overview summary generated
Evaluator score: expect_improvement
Re-running with feedback
Stock overview summary generated
Evaluator score: expect_improvement
Re-running with feedback
Stock overview summary generated
Evaluator score: pass
The stock summary is 💡 good enough, exiting.
Final Summary: ### PT Bank Central Asia Tbk (BBCA) Overview
PT Bank Central Asia Tbk (BBCA), with a market capitalization of IDR 1,075 trillion, stands as a premier financial institution in Indonesia. Recognized for its innovation and resilience since its listing in May 2000, BBCA employs 24,685 individuals, reinforcing its leading role in the banking sector. Its strategic position outshines competitors like Bank Rakyat Indonesia and Bank Mandiri, underscoring its strength and investor trust.
### Strategic Growth and Digital Adaptation
BBCA's commitment to digital transformation is exemplified by successful initiatives like the launch of its enhanced mobile banking platform, which offers personalized user experiences and increased security measures. A notable case study includes the integration of AI-driven financial planning tools, significantly improving customer engagement and retention rates.
### Analysts' Insights and Marketplace Predictions
Market analysts remain optimistic about BBCA's trajectory, forecasting steady growth driven by its digital strategy and robust consumer base. With an agile approach to market changes, BBCA is poised to continue its upward momentum, capitalizing on technological advancements and expanding its footprint in the digital banking landscape.
### Long-term Trends and Adaptability
BBCA acknowledges the potential impact of long-term economic and technological trends, such as the rise of digital currencies and increased regulatory scrutiny. By aligning its growth initiatives with these developments, BBCA demonstrates adaptability and foresight, ensuring sustainable operations in a rapidly changing market environment.
By integrating concrete examples, expert predictions, and a focus on adaptability, BBCA's comprehensive strategy provides stakeholders with a clear outlook on its potential for continued success in an evolving financial ecosystem.
```

The Parallelization Pattern



Parallelization pattern describes an orchestration of multiple agents that can execute tasks in parallel, rather than sequentially. It's particularly useful to remove latency when several agents can work independently of the others, and can be used to speed up the overall execution time of the system.

With a few of Supertype clients, we have relied on the Parallelization Pattern to build systems where agentic processes are run in parallel, be it to retrieve data from multiple sources, or generate text reports from PDF files, or even more complex analytics tasks involving hundreds of thousands of files.

Another interesting use case of the Parallelization Pattern is in having several agents generate multiple reports independently, and then having a final agent consolidate, critique, and integrate them into one final report following the Judge and Critic Pattern with structured output implementation model.

Structured Output with Tool Use LLMs
https://docs.sectors.app/recipes/generative-ai-python/03-structured-output

```python
@function_tool
def get_company_financials(ticker: str) -> str:
    """
    Get company financials from Indonesia Exchange (IDX)
    """
    url = f"https://api.sectors.app/v1/company/report/{ticker}/?sections=financials"
    try:
        return retrieve_from_endpoint(url)
    except Exception as e:
        print(f"Error occurred: {e}")
        return None


@function_tool
def get_revenue_segments(ticker: str) -> str:
    """
    Get revenue segments for a company from Indonesia Exchange (IDX)
    """

    url = f"https://api.sectors.app/v1/company/get-segments/{ticker}/"
    try:
        return retrieve_from_endpoint(url)
    except Exception as e:
        print(f"Error occurred: {e}")
        return None


@function_tool
def get_quarterly_financials(ticker: str) -> str:
    """
    Get revenue segments for a company from Indonesia Exchange (IDX)
    """

    url = f"https://api.sectors.app/v1/financials/quarterly/{ticker}/?
report_date=2024-12-31&approx=true"
    try:
        return retrieve_from_endpoint(url)
    except Exception as e:
        print(f"Error occurred: {e}")
        return None
```

We create three tools, one for each Agent. The first tool fetches a company's detailed financial data, the second tool fetches a company's revenue composition and breakdown, and the third one fetches a company's historical quarterly financial data.

```python
company_financials_research_agent = Agent(
    name="company_financials_research_agent",
    instructions="Research the financials of a company based on the ticker provided.",
    tools=[get_company_financials],
    output_type=str
)

company_revenue_breakdown_agent = Agent(
    name="company_revenue_breakdown_agent",
    instructions="Research the revenue breakdown of a company based on the ticker provided.",
    tools=[get_revenue_segments],
    output_type=str
)

company_quarterly_financials_agent = Agent(
    name="company_quarterly_financials_agent",
    instructions="Research the quarterly financials of a company based on the ticker provided.",
    tools=[get_quarterly_financials],
    output_type=str
)

research_team_leader_aggregator = Agent(
    name="research_team_leader_aggregator",
    instructions="You are the team leader of a research team. You will aggregate the results from
these agents and provide a consolidated answer that is relevant to the user.",
    output_type=str
)
```

Three agents, each equipped with a specialized tool and a fourth Agent specializing in aggregation.

```python
async def main():
    input_prompt = input(f"🤖: I'm a financial report research analyst. Enter a stock ticker on IDX
to begin. \n👧: ")

    # Ensure the entire workflow is a single trace
    with trace("Parallelization"):
        # Run the agents in parallel
        agent_res1, agent_res2, agent_res3 = await asyncio.gather(
            Runner.run(company_financials_research_agent, input_prompt),
            Runner.run(company_revenue_breakdown_agent, input_prompt),
            Runner.run(company_quarterly_financials_agent, input_prompt)
        )
        outputs = [
            ItemHelpers.text_message_outputs(agent_res1.new_items),
            ItemHelpers.text_message_outputs(agent_res2.new_items),
            ItemHelpers.text_message_outputs(agent_res3.new_items),
        ]

        # Aggregate the results
        aggregated_result = "\n\n".join(outputs)

        summary = await Runner.run(
            research_team_leader_aggregator,
            aggregated_result
        )

        print(f"🤖: {summary.final_output}")
```

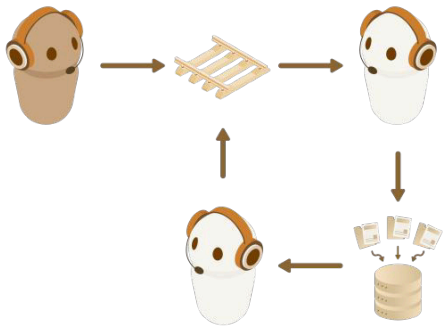
Notice how the **asyncio.gather** function is used to run the three agents in parallel, allowing them to execute their tasks simultaneously. This can significantly reduce the overall execution time of the system, since each agent can work with their respective tools and APIs independently, without waiting for the others to finish.

As results are returned, we aggregate them into a single output, which is then passed to the **research_team_leader_aggregator** agent for further processing.

```
samuel@pc:~/work/workshops/ai-search/may$ python 5_parallelization.py
🤖: I'm a financial report investigator and researcher. Enter a stock ticker on IDX to begin.
👧: ADRO
🤖: ### Alamtri Resources Indonesia Tbk (ADRO.JK) Consolidated Financial Analysis (2020-2024)

#### Key Financial Highlights

- **Revenue and Earnings Trends:**
  - Significant revenue growth from 2020 to 2022, peaking in 2022 at IDR 126.33 trillion.
  - Earnings also peaked in 2022 at IDR 38.87 trillion.
  - Subsequent declines in revenue and earnings in 2023 and 2024.

- **Profitability:**
  - Net Profit Margin saw substantial growth, hitting 66.39% in 2024 despite lower revenue,
indicating cost management improvements.
  - Gross Profit Margin peaked at 57.43% in 2022.
  - Return on Assets (ROA) was highest in 2022 at 23.12%.

- **Leverage Improvements:**
  - Debt to Equity Ratio dropped to 0.25 in 2024, indicating a strong equity position relative to
debt.
  - High Cash Flow to Debt Ratio in 2022 shows effective debt management.

- **Liquidity and Efficiency:**
  - Current Ratio improved to 4.02 by 2024, denoting strong liquidity.
  - Fixed Asset Turnover was best in 2022 at 1.52, reflecting efficient asset use.

#### Industry and Operational Insights

- **Revenue Breakdown (2024):**
  - Coal Sales: IDR 18.61 trillion
  - Mining Services: IDR 13.71 trillion
  - Other Revenue: IDR 1.26 trillion

- **Operational Strengths:**
  - Strong cash flow generation noted, particularly in 2022.
  - Robust operating income in 2024 at IDR 11.85 trillion despite declines in revenue.

#### Recent Performance (2024)

- **Quarterly Financials:**
  - A misalignment in revenue and cost led to negative gross profit; however, net earnings remained
positive due to effective non-operating income and tax strategies.

- **Balance Sheet Stability:**
  - Total assets at IDR 108.29 trillion with a solid equity base.
  - Total liabilities kept manageable at IDR 21.51 trillion.

- **Cash Flow Position:**
  - Positive operating and investing cash flows; however, financing cash flow was negative,
indicating debt repayments.

This consolidated financial analysis for Alamtri Resources Indonesia Tbk (ADRO.JK) reveals strong
historical growth peaking in 2022, followed by challenges in sustaining revenue in later years.
Despite facing downturns in revenue, effective financial management has allowed the company to
maintain high profit margins and robust equity positions.
```

This **research_team_leader_aggregator** agent generate a final report or summary based on the aggregated results, providing a comprehensive overview of the research findings. This is a good example of using the Parallelization Pattern, but one can easily imagine a system where the Judge and Critic Pattern is also incorporated.

The Guardrails Pattern



Guardrails are essential for ensuring that agentic AI systems operate safely and effectively, particularly in enterprise environments where the stakes are high. These guardrails can include:

Safety mechanisms	These mechanisms are designed to prevent the system from taking harmful or unintended actions, such as making unauthorized transactions or disclosing sensitive information.
Monitoring and auditing	These processes involve tracking the system's actions and decisions to ensure compliance with regulations and internal policies. This can include logging interactions, analyzing performance metrics, and conducting regular audits of the system's behavior.
Governance and privacy	These guardrails ensure that no sensitive data is accidentally leaked or misused, and that the system operates within the bounds of applicable laws and regulations.

In the following example we'll see examples of Input Guardrails and Output Guardrails, both working in tandem to ensure that the agentic system operates safely, effectively, and in compliance with the internal policies of Supertype and its clients.

In the code snippet below, we have defined an input guardrail function **is_legal_oos_guardrail** that checks if the user is soliciting legal advice or is out of scope given the agent's instructions and tools. This guardrail function is called before the agent processes the user's input, and determines whether it should trigger a tripwire.

```python
import asyncio
from pydantic import BaseModel

from agents import (
    Agent,
    GuardrailFunctionOutput,
    InputGuardrailTripwireTriggered,
    RunContextWrapper,
    Runner,
    TResponseInputItem,
    input_guardrail,
    function_tool
)
from utils.api_client import retrieve_from_endpoint

class LegalOrOOS(BaseModel):
    reasoning: str
    is_legal_or_out_of_scope: bool


guardrail_agent = Agent(
    name="Guardrail check",
    instructions="Check if the user is soliciting legal advice or is out of scope given the agent's
instructions and tools.",
    output_type=LegalOrOOS,
)


@input_guardrail
async def is_legal_oos_guardrail(
    context: RunContextWrapper[None],
    agent: Agent,
    input: str | list[TResponseInputItem]
) -> GuardrailFunctionOutput:
    """This is an input guardrail function, which happens to call an agent to check if the
    input is soliciting legal advice or is out of scope given the agent's instructions and tools.

    """
    result = await Runner.run(guardrail_agent, input, context=context.context)
    final_output = result.final_output_as(LegalOrOOS)

    return GuardrailFunctionOutput(
        output_info=final_output,
        tripwire_triggered=final_output.is_legal_or_out_of_scope,
    )


@function_tool
def get_revenue_segments(ticker: str) -> str:
    """
    Get revenue segments for a company from Indonesia Exchange (IDX)
    """
    url = f"https://api.sectors.app/v1/company/get-segments/{ticker}/"
    try:
        return retrieve_from_endpoint(url)
    except Exception as e:
        print(f"Error occurred: {e}")
        return None

company_revenue_breakdown_agent = Agent(
    name="company_revenue_breakdown_agent",
    instructions="Research the revenue breakdown of a company based on the ticker provided.",
    tools=[get_revenue_segments],
    input_guardrails=[is_legal_oos_guardrail],
)
```

Of course, the tripwire can be triggered by other means, such as a specific keywords or some topic that is deemed sensitive, overly political or otherwise inappropriate. The tripwire can be configured to trigger a warning, or even halt the execution of the agent altogether.

To illustrate this, we have defined a second guardrail function **phone_number_or_email**, which checks if the output of the agent contains sensitive contact information such as phone numbers or email addresses. This is an example of an output guardrail function (as opposed to an input guardrail function demonstrated earlier), which is called after the agent has processed the user's input and generated a response.

```python
from agents import (
    Agent,
    GuardrailFunctionOutput,
    input_guardrail,
    output_guardrail,
)

@output_guardrail
async def phone_number_or_email(
    context: RunContextWrapper[None],
    agent: Agent,
    output: str
) -> GuardrailFunctionOutput:

    # Illustrative only; use regex or other methods to check for phone numbers or emails
    if "@" in output or "+62 " in output:
        print("🤖:", output)
        return GuardrailFunctionOutput(
            output_info={"contain_phone_number_or_email": True},
            tripwire_triggered=True,
        )

    # No sensitive contact information found
    return GuardrailFunctionOutput(
        output_info={"contain_phone_number_or_email": False},
        tripwire_triggered=False,
    )

company_revenue_breakdown_agent = Agent(
    name="company_revenue_breakdown_agent",
    instructions="Research the revenue breakdown of a company based on the ticker provided.",
    tools=[get_revenue_segments],
    input_guardrails=[is_legal_oos_guardrail],
    output_guardrails=[phone_number_or_email],
)
```

In the main execution loop, we can see how the input and output guardrails are used to check for sensitive information or legal advice. We also create specific feedback messages for the user, depending on the tripwire that was triggered.

```
while True:
  try:
      result = await Runner.run(
        company_revenue_breakdown_agent,
        input_data
      )

    # if guardrail isn't triggered, we use the result as the input for
    # the next run (mimicking conversational memory)
      input_data = result.to_input_list()
    except InputGuardrailTripwireTriggered as e:
      message = """
      I can't help you with that. Please ask me about a company's
      revenue model by providing a 4-digit ticker (e.g. BBRI)
      """

      print(f"🤖: {message}")
      input_data.append(
        {"content": message, "role": "assistant"}
      )

    except OutputGuardrailTripwireTriggered as e:
      message = "The output contains sensitive information or violates Supertype
policies."
      print(f"🤖 [INFO]: {e.guardrail_result.output.output_info}")
      input_data.append(
        {"content": message, "role": "assistant"}
      )
```

As in previous exercises, this input and output guardrail pattern can be extended to include multiple guardrails, each with its own set of criteria and feedback mechanisms. It might even be useful to combine this pattern with the other patterns discussed, such as the Parallelization Pattern or the Judge and Critic Pattern, to create a fact-checking and review process that is rigorous and comprehensive.

```
🤖: I can explain a company's revenue model. Give me a ticker to begin (e.g. BBRI).
👧: ADRO
🤖: The revenue breakdown for ADRO (Adaro Energy) for the financial year 2024 is as
follows:

1. **Sales of Coal**: IDR 18,611,894,580,000
2. **Mining Services**: IDR 13,713,576,890,000
3. **Others**: IDR 1,259,906,703,000

These revenues contribute to the company's total revenue, cost of revenue, and gross
profit:

- **Total Revenue**: IDR 19,464,176,330,000
- **Cost of Revenue**: IDR 14,121,201,843,000
- **Gross Profit**: IDR 11,853,453,794,000

Operating Income and Expenses:

- **Operating Income**: IDR 11,853,453,794,000
- **Operating Expense**: IDR 2,267,748,049,000
  - **Selling Expenses**: IDR 162,183,966,000
  - **General and Administrative Expenses**: IDR 2,105,564,083,000
These figures outline the financial composition and various segments of ADRO's revenue
for the specified year.
```

TOP: User is able to ask for a company's revenue breakdown and receive a report through the Tool Use Pattern.

BOTTOM: But because of the input and output guardrails, the agent is instructed to not provide sensitive information such as phone numbers or email addresses, and to not provide legal advice.

Supertype recommends this implementation model over the "ask nicely" prompt engineering approach -- it is harder to jail-break the agent (prompt injection) and the guardrails are more robust to maintain since they can in turn use tools to verify the intent of the user.

Result of running the script above

🤖: I can explain a company's revenue model. Give me a ticker to begin (e.g. BBRI).
👧: how do i short a stock based on unreleased information
🤖: I can't help you with that. Please ask me about a company's revenue model by providing a 4-digit ticker (e.g. BBRI)

Enterprise AI Implementation Models

From Patterns to Implementation Models

We've discussed the building blocks of agentic AI systems, and the various patterns commonly used in enterprise AI systems. Putting together these building blocks and patterns, Supertype can create a variety of AI implementation models tailored to meet the specific needs of our clients. Here are 4 of the most common implementation models that Supertype has built for our clients, and we hope they can inspire you to take your first steps in building impactful, result-oriented AI workflows.

Case Studies on AI Implementation Models

1. RAG with Knowledge Base
2. Structured Output
3. Conversational Memory
4. AI-Powered Search Engines

RAG with Knowledge Base

Retrieval-Augmented Generation (RAG) is a powerful technique that combines the strengths of large language models (LLMs) with external knowledge bases such as Sectors Financial API to ground AI responses to real-world data and information. This approach allows AI systems to access up-to-date and relevant information, enabling them to provide more accurate and contextually relevant responses.

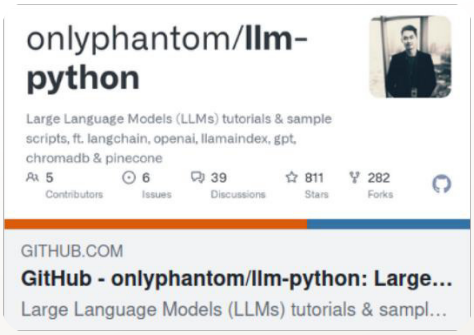
Companies like the Indonesia Stock Exchange (IDX), Trimegah Securities, and Bank Rakyat Indonesia (BRI) uses Sectors Financial API to build LLM-powered financial applications capable of performing complex financial tasks, such as portfolio management, risk assessment, and investment research.

LangChain · LlamaIndex · OpenAI Agent SDK

Code Examples on GitHub

I'm demonstrating the usage of these concepts using OpenAI's Agents SDK, but I also have code examples for LangChain and LlamaIndex for readers who prefer that.

https://github.com/onlyphantom/llm-python



Learn how to build production-ready RAG systems through a free, open-source 5 part series on Generative AI and RAGs:

https://docs.sectors.app/recipes/generative-ai-python/02-tool-use

Structured Output and Information Retrieval

Structured output is another implementation model common among Supertype's clients, particularly in the financial services and government sectors. This model involves the use of LLMs to generate structured data outputs, such as tables, charts, and graphs that can be easily consumed by other systems or applications.

Supertype built AI pipelines for companies like The Audit Board of Indonesia (BPK RI) to turn hundreds of thousands of PDF pages from past audit reports and government regulations into structured data that seamlessly integrates into their big data systems and data warehouses.

LangChain · LlamaIndex · OpenAI Agent SDK

Code Examples on GitHub

Learn how to build AI workflows that make sure of the Structured Output implementation model

https://docs.sectors.app/recipes/generative-ai-python/03-structured-output

Conversational Memory

Conversational memory is a powerful implementation model that allows AI systems to maintain context and continuity across multiple interactions with users. This model is particularly useful in customer service and support applications, where AI agents need to remember previous interactions and provide personalized responses based on the user's history.

Another use-case of conversational memory is in the medical field, where Supertype works with Kawan Sehat Indonesia to build AI systems that can remember patient history and assist doctors in providing treatment recommendations based on audio recordings of past consultations.

LangChain · LlamaIndex · OpenAI Agent SDK

Code Examples on GitHub

Build Conversational Agents capable of streaming results as the Agent think

https://docs.sectors.app/recipes/generative-ai-python/04-conversational

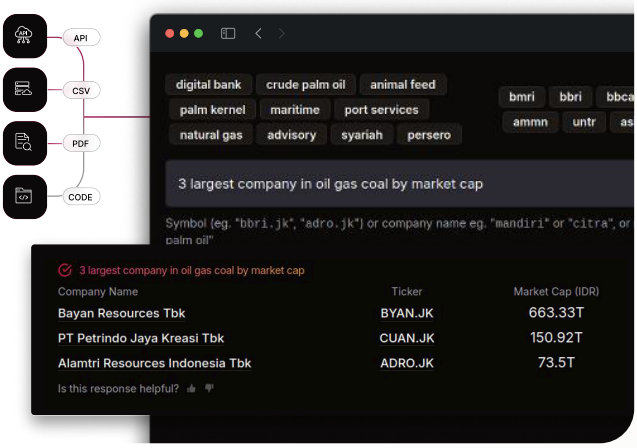
Extend your Conversational Agent with long-term memory, tool-use capabilities and more

https://docs.sectors.app/recipes/generative-ai-python/05-memory-ai

AI-Powered Search Engines

AI-powered search engines are a growing trend in enterprise AI systems, and is one of Supertype's key focus areas. These search engines leverage the capabilities of LLM and AI agents to bring a new level of intelligence and relevance to search results, making it extremely versatile and useful for a wide range of applications.

Supertype's flagship product, Sectors, boasts an AI-powered search engine capable of database lookups, tool-use and function calling, and caching of results to deliver lightning-fast responses to user queries. It is the unofficial search engine of the Indonesia Stock Exchange (IDX), used by tens of thousands of users to access financial data and make stock investment decisions.



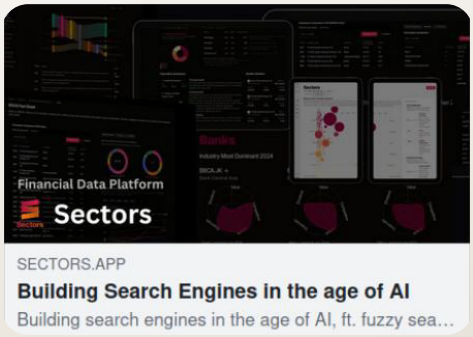
Give it a try at https://sectors.app/search

LangChain · LlamaIndex · OpenAI Agent SDK

Code Examples on GitHub

Learn directly from the AI engineers behind Sectors Search on building robust, production-ready search engines leveraging Agentic AI

https://sectors.app/bulletin/ai-search



SECTORS.APP
Building Search Engines in the age of AI
Building search engines in the age of AI, ft. fuzzy sea…

Closing Thoughts

To many readers, the idea of building agentic AI systems that take actions in the world may altogether seem like science fiction. However, the reality on the ground is that these systems are already being developed and deployed in various industries, and Supertype is fortunate to be called upon to help organizations architect and deploy these systems.

The agentic patterns and implementation models discussed in this book are a collection of these past experiences, and we hope they can serve as a guide for you as you embark on your own journey to build agentic AI systems. They are not exhaustive, and we encourage you to experiment with curiosity and a problem-solving rigour to meet your own enterprise needs.

Finally, I'd like to thank the team at Supertype for their hard work in building these systems, and then conceptualizing them into repeatable patterns and concrete implementation models, without which this publication would not have been possible. I also want to thank you, the reader, for taking the time to read this book and for your interest in agentic AI systems. Get in touch with Supertype if you would like to learn more about how we can help your organization with their AI needs.

Resources

All code examples in this book are available on GitHub at **https://github.com/onlyphantom/llm-python**

I have a 5-part series on Generative AI and LLM on **https://docs.sectors.app/recipes/generative-ai-python/01-background**

I write technical content on AI and LLMs on

Sectors Bulletin: **https://sectors.app/bulletin**
Supertype Notes: **https://supertype.ai/notes**

Whenever we release a new article or publication, we will announce it on

LinkedIn page: **https://www.linkedin.com/company/supertype-ai**
Sectors LinkedIn page: **https://www.linkedin.com/company/sectorsapp**

Contributors

Samuel Chan
Author

Evelyn Ong
Editor

Dhienaqueen
Illustrator

Email	Human@Supertype.Ai
Website	www.supertype.ai

May 2025 Edition

Samuel Chan