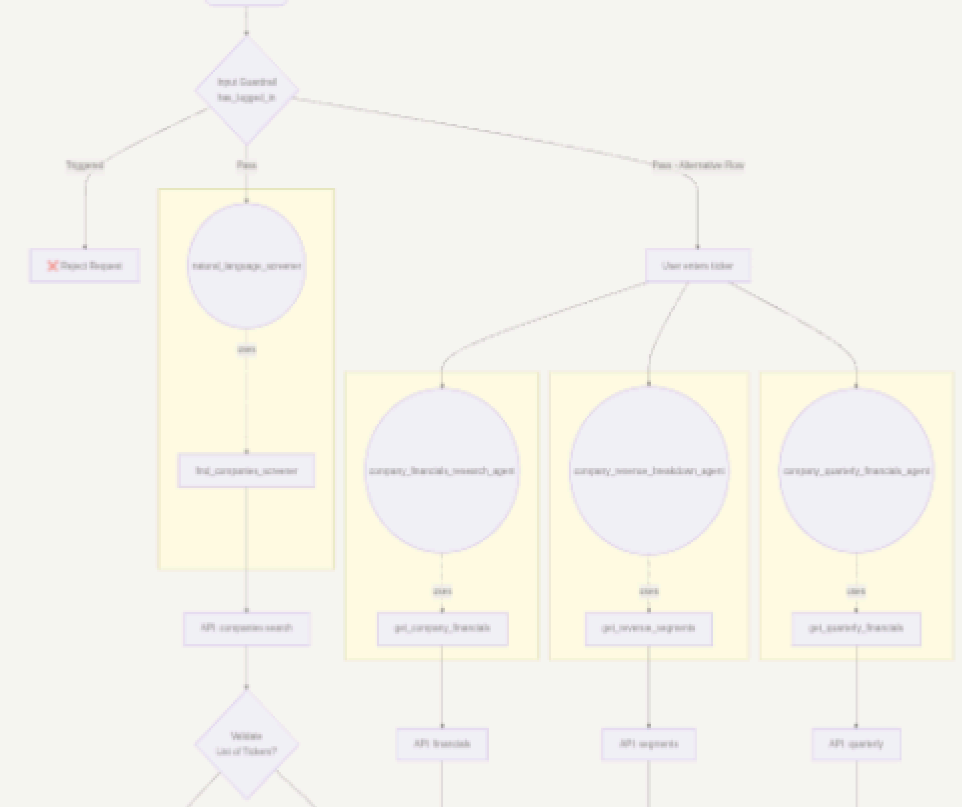


Blueprints from Agentic Patterns

The 6 Agentic AI Patterns for orchestrating Multi-Agent Enterprise Systems

Preview Edition

Supertype Publication

Samuel Chan

Feb 2026

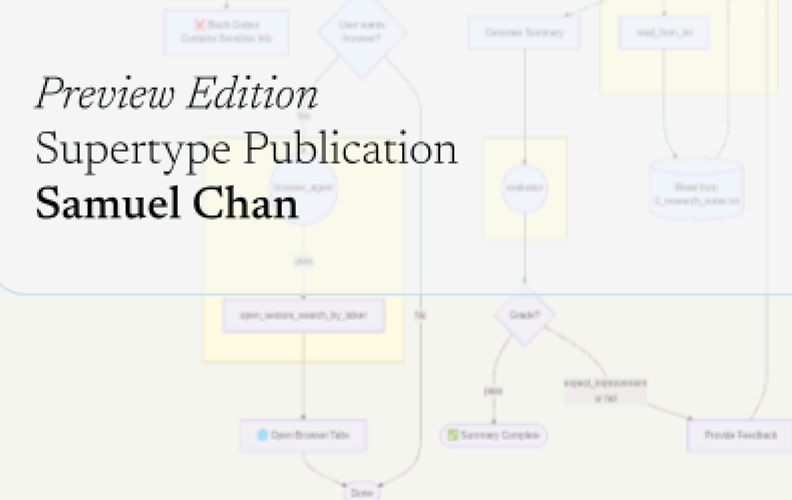


Table of Contents

Table of Contents 2

Introduction 3

Foreword 3

About the Author 5

Using non-OpenAI Models with this book 6

Glossary and Common Terminology 8

AI Agents for Enterprises 11

Enterprise Considerations 12

A Note on Hallucinations 14

Architecting against Hallucinations 15

The Problem with Just Prompt Engineering 16

Building Agentic Systems 19

Agent Foundations 19

Stepping up to Multi-Agent Systems 21

6 Agentic AI Patterns 24

The Manager Pattern 26

A Small Variation: The Hand-Off Pattern 28

The Skills and Tools Pattern 29

Creating a Tool for an AI Agent 30

Tool Use Behavior by Agents 33

The Sequential Chain Pattern 35

Agentic Personal Assistants 43

The Judge and Critic Pattern 45

The Parallelization Pattern 50

Case Study: Sectors Financial Data Platform 50

Building a Multi-Agent Stock Research Program 51

The Guardrails Pattern 58

Putting it all together: The Blueprint 64

Where to go from here 66

Other Resources 67

Introduction

🔖 Chapter Objectives

Understand the motivation behind this book, its structure, and how to use it effectively. Also, familiarize yourself with common terminology and key industry terms that will be used throughout the book.

Foreword

03 Feb 2026

Following the release of the Agentic Patterns book in May 2025, my team and I at Supertype have been refining our approach to building high-scale, enterprise-grade AI solutions for our clients. We have, accordingly, developed a set of blueprints that codify our learnings from these projects, and are releasing the Blueprints for Enterprise AI as a way to disseminate these learnings to our AI engineering teams at Supertype and secondly, to the broader community of AI practitioners and key stakeholders.

These blueprints are designed to provide a structured approach to building AI solutions that are robust, scalable, and aligned with business objectives. They encapsulate best practices, architectural patterns, and implementation strategies that we have found effective in our work.

Through various past engagements, as well as the high-stakes undertaking of building Southeast Asia's first financial AI suite, I have identified and documented a set of successful strategies and common mental models that we employ when architecting agentic AI solutions -- we use these blueprints as a reference point when scoping, designing or communicating about AI solutions with our clients. Sectors, our financial AI Suite, counts companies like Trimegah Sekuritas, Mandiri Sekuritas, and many other companies as annual subscribers; our AI-as-an-API service powers mission-critical workflows for more than 30,000 users

monthly; and our Sectors AI Search and Sectors AI Chat products regularly come up top in accuracy and speed benchmarks against similar products from global tech giants.

Here is a book that documents these blueprints, and the underlying patterns that we use to build them. There will be accompanying code examples, plenty of architectural diagrams, and real-world examples to illustrate the concepts discussed. If you are not a technical user of AI -- these Blueprints can still be well-understood without necessarily diving into each line of code.

This is Book 1 of 2 in the Enterprise AI series. Book 2, titled *The Enterprise AI Agentic Stack*, focuses on the technological deliverables and enterprise stack required to materialize these Blueprints into AI products and services.

Book	**Key Themes**	**Frameworks**
1 Blueprints from Agentic Patterns	The 6 Agentic AI Patterns for orchestrating Multi-Agent Enterprise Systems	📐 6 Agentic Patterns
2 The Enterprise AI Agentic Stack	Building and Managing Agentic Systems using established Blueprints	🌿 LEAVES Deliverables

It is my hope that both these books together will serve as a valuable resource for any leadership, product management, or business stakeholders tasked with equipping their organizations with the best chance of success in their AI initiatives.

Best,
Samuel (Head of Supertype Incubator)

About the Author

Samuel is the founder of Algoritma, Indonesia's most acclaimed data science school trusted by more than 500 companies and institutions across Indonesia, Singapore and Thailand. He has built and managed numerous software and analytics development teams for businesses across Asia Pacific since 2014, and has built and sold multiple successful software automation businesses whilst living between Beijing, Tokyo and home town Singapore.

He is also the founder of Supertype, an AI Development and Analytics Engineering consultancy trusted by Indonesia's largest enterprises. Under his capacity as Head of Incubator, the company develops and runs the region's leading LLM-first financial data platform, built to serve thousands of companies and developers building AI Agents grounded on real world data.

Samuel is also an accredited instructor with more than 12 years of enterprise consulting experience and the qualifications to teach at Ngee Ann Polytechnic, IBF (Institute of Banking and Finance) Singapore, NTUC LearningHub Singapore, and various other institutions. He is also a recognized expert in Large Language Models (LLMs), with more than 10,000+ subscribers and more than 250,000 views on his LLM Generative AI tutorial series on YouTube, and a contributor to various publications and research bodies including The Logistics Institute by NUS (National University of Singapore). Prior to founding Supertype, Samuel was a three-time software entrepreneur and an in-house consultant to SEGA (TYO: 6460) subsidiaries, Alamtri (ADRO) Group, DNP Group (TYO: 7912), Gumi (TYO: 3903), DianDian FunPlus (SH:600634) etc. during his 18-year career – building analytics and software teams for regional-leading enterprises. Notably, Samuel is also the first recipient of RStudio (now Posit) Certified Trainer status in Asia.

Using non-OpenAI Models with this book

The code examples provided in this book are largely based on OpenAI's Agents SDK, which is a pedagogical choice to provide a python-centric, easy-to-follow example for readers to get started with. However, the architectural patterns and concepts discussed in this book are not tied to any specific LLM provider or framework. You can refer to my open-source LLM-in-Python repository where you will find more than 40+ tutorial videos and Python scripts that are centered around LangChain, LlamaIndex, Pinecone, and other popular frameworks -- all of which can be used to implement the same Agentic AI Patterns and Blueprints discussed in this book.

To make the most out of it, you need only install the openai-agents library (along with its dependencies) in your Python environment. You can do this via pip:

```
pip install openai-agents
```

To use other non-OpenAI models, install the litellm dependency group:

```
pip install openai-agents[litellm]
```

The rest of the code is identical with the small substitution of the model name to your desired LLM provider with the litellm prefix:

```
claude_agent = Agent(
  model="litellm/anthropic/claude-3-5-sonnet-20240620", ...)
gemini_agent = Agent(
  model="litellm/gemini/gemini-2.5-flash-preview-04-17", ...)
```

You can also set a specific model for all Agents globally by exporting an environment variable at the start of your session:

```
export OPENAI_DEFAULT_MODEL=gpt-4.1
```

The gpt-4.1 family (including the mini and nano variants) are recommended for this book, as they provide a good balance between performance and cost. You do have the choice of using a GPT-5.x model, along with other models from Anthropic, Google, and Meta -- but be aware that the performance characteristics and cost implications may vary.

```
financial_assistant = Agent(
    name="Financial Assistant",
    instructions="Help me make the most out of my investment decisions",
    model="gpt-5.2",
    # optional since it has reasonable default,
    # but can be used to change its reasoning effort
           model_settings=ModelSettings(reasoning=Reasoning(effort="high"),
verbosity="low")
)
```

Glossary and Common Terminology

Generative AI is a subset of artificial intelligence that focuses on creating new, original content by learning from existing data. This is in contrast to more conventional AI applications where the task is to predict or classify data.

A subset of Generative AI that is gaining popularity is the use of **Large Language Models** (LLMs). These are models trained on large amounts of text data — such as books, articles, websites and online forums — to generate human-sounding text. A common example of an LLM is Meta's Llama models, which have been used to generate everything from poetry to writing code. Outside of text generation, the same principles that underlie LLMs can be applied to other types of data, such as images and music, opening a new frontier of possibilities for creative applications.

In some cases, the model is fed with such an enormous amount of data across both textual and non-textual data that it can support text-generation tasks but also understand images and other forms of data. These types of models are referred to as **multimodal models**.

When Supertype undertakes AI engineering initiatives in the past, we sometimes introduce a phase known as **fine-tuning**. Fine-tuning is the process of adapting a pre-trained large language model (like the aforementioned Llama) for a specific task – financial data retrieval, advisory-style Q&A capabilities, wiki generation are examples that spring to mind.

We do this by exposing the base model on specially chosen training examples, or by continuing its training on smaller domain-specific datasets, or by augmenting it with our data from our proprietary Sectors Financial Data Suite so the AI models we train have highly specific, timely knowledge.

When a Large Language Model is paired with the ability to retrieve and generate answers based on some external, trusted data store, it satisfies the requirements of a **Retrieval Augmented Generation (RAG)** implementation. By retrieving data from this external, trusted data store – such as proprietary company documents, or databases, or financial APIs – the generated response is grounded to some verified, up-to-date information.

💡 Tips Alternatives to Retrieval Augmented Generation?

At Supertype, we often explain to our clients how RAG is an architecture conceived out of the need to address two problems with LLM-only AI systems: (1) high rate of **hallucination** and (2) a **knowledge cutoff** problem, where the LLM is unable to answer questions outside of its training data.

By pairing the typical generative capabilities of a LLM with a retrieval system and calibrated system prompts, we greatly improve the model's ability to generate responses grounded on actual historical data – and thus delivering trustworthy information that are as up to date as the external knowledge store, or stores, powering it. But RAG *doesn't* have to be the only solution. Can you think of a theoretical alternative to addressing the two problems above?

Specifically for books in the Enterprise AI series, we shall also refer to the following terms frequently. Conveniently, the 4 terms also form an acronym: ABCD, which you can use to remember them by.

Whenever they are referred to in the book, they will be capitalized to provide a hint that they should be interpreted as contextualized terms within the Enterprise AI series.

Book	Definition
Agent	An AI Agent that leverages LLMs to perform specific tasks autonomously or semi-autonomously.
Blueprint	An architectural design built from the Agents and Patterns, outlining how these components interact to achieve specific enterprise AI objectives.
Considerations	A set of unique requirements and challenges that enterprises must address when deploying AI Agents, detailed in Enterprise Considerations.
Deliverable	One of the LEAVES pillars that group tasks-to-be-done into functionally distinct, yet interconnected pillars for materializing Blueprints.

AI Agents for Enterprises

🔖 Chapter Objectives

Understanding what makes an AI 'Agentic', the unique considerations for Enterprises when deploying AI Agents, and how to engineer against inaccuracies and hallucinations.

Large Language Models (LLMs) refer to a class of AI models that can understand and generate human-like text based on the data they have been trained on. At the lowest level, they are sophisticated pattern recognition systems that predict the next word in a sequence based on the context provided by previous words. Modern foundation models like GPT-4, Claude, and Llama 3 are examples of LLMs that have been trained on vast amounts of text data from the internet, books, articles, and other sources.

When we talk about **AI Agents**, we refer to systems that leverage LLMs to perform specific tasks autonomously or semi-autonomously. These agents can interact with users, process information, and make decisions based on the data they have been trained on and the context of their interactions. AI Agents can be designed to handle a wide range of applications, from customer service chatbots, to Excel-filing automations, decision support systems, SOP compliance checkers, and more. These action-oriented capabilities distinguish AI Agents from traditional LLM applications that primarily focus on text generation.

Enterprise Considerations

When deploying AI Agents in an enterprise context, several unique considerations come into play:

1. Data Security and Privacy
   Enterprises often deal with sensitive data, making it crucial to ensure that AI Agents comply with data protection regulations and internal security policies.

2. Integration with Existing Systems
   AI Agents need to seamlessly integrate with existing enterprise systems and workflows to provide value without disrupting operations. Commonly, this involves connecting with Enterprise Resource Planning (ERP) systems (e.g. Odoo or SAP HANA), CRM platforms, documents and other internal tools.

3. Scalability
   Enterprises require AI processes that can scale to handle large volumes of data processing especially during peak times or high-demand scenarios.

4. Customization
   AI Agents must be tailored to meet the specific needs and requirements of the enterprise, including industry-specific knowledge and company-specific processes.

5. Governance and Compliance
   Enterprises must ensure that AI Agents adhere to regulatory requirements and internal governance policies. By way of example, financial companies in Indonesia must ensure that their processes comply with OJK regulations -- this extends to the data handling and processing performed by AI Agents as well.

6. **User Experience**
   Enterprise users are distinct from general consumers, and AI Agents must be designed to provide a user experience that meets the expectations and needs of enterprise users. In *The Enterprise AI Agentic Stack* book, we shall discuss the Enterprise Experience deliverable of LEAVES in greater detail.

7. **Trustworthiness**
   Confidence is paramount in enterprise adoptions of AI agents. The agents must be able to substantiate their outputs with verifiable data sources, and provide transparency into their recommendations and decision-making processes. Whenever possible, answers need to be traceable to specific data points or documents.

These seven points, along with the addition of mitigating hallucinations (discussed in the next sub-section), form the core Considerations for Enterprise AI (**Considerations**) heavily referenced throughout this book. They serve as guiding principles when architecting resilient Agentic AI blueprints that meet the stringent requirements of enterprise deployments.

A Note on Hallucinations

One of the most significant challenges with LLMs (of which AI Agents are built upon) is the phenomenon of **hallucinations**. Hallucinations occur when the model generates information that is not based on its training data or any real-world knowledge, leading to outputs that are factually incorrect or plausible-sounding but entirely fabricated.

In enterprise applications, hallucinations can have serious consequences, such as providing incorrect financial advice, misinterpreting legal documents, or generating misleading customer communications.

The above **Considerations** and the challenge of Hallucinations underscore the importance of adopting robust architectural patterns and best practices when building AI Agents for enterprise use cases. The subsequent sections will delve into the specific Agentic AI Patterns that address these challenges and enable the development of effective, reliable, and scalable AI solutions for enterprises.

Architecting against Hallucinations

To mitigate the risk of hallucinations in AI Agents, several architectural strategies can be employed:

1. Retrieval-Augmented Generation (RAG)
   By integrating a retrieval system that sources information from trusted databases or documents, AI Agents can ground their responses in verified data, reducing the likelihood of hallucinations.
2. Human-in-the-Loop (HITL)
   Incorporating human oversight into the AI Agent's workflow allows for real-time validation and correction of outputs, ensuring that any potential hallucinations are caught and addressed before they impact users.
3. Logs and Tracing
   Implementing systems to monitor the performance of AI Agents and gather user feedback can help identify patterns of hallucinations and inform ongoing improvements to the model and its architecture. Logs and Tracing is one of LEAVES Deliverables discussed in *The Enterprise AI Agentic Stack* book.

Notice that these strategies align closely with the Considerations outlined earlier, emphasizing the need for trustworthiness, governance, and user experience in enterprise AI applications. Also worth noting is that "prompt engineering" itself – while important – is insufficient on its own to fully address hallucinations without these broader architectural Considerations in place.

The Problem with Just Prompt Engineering

Prompt engineering describes the process of crafting and refining the input prompts (much like an instruction to the AI model) to elicit a desired response from a large language model (LLM). An example of prompt engineering might involve specifying the desired format of the response, the tone (e.g., formal or casual), or asking the model to focus on certain aspects of a topic to suit a particular use case.

Consider a sample prompt engineering instruction like the following:

> *When answering, please provide a concise summary after gathering all relevant information from the 4 specified trusted documents. Use English only, even if the source documents are in Bahasa Indonesia or other languages. Do not make any legal recommendations, and do not access the* `crm_03` *and* `crm_05` *tables. If you need to verify any financial numbers, use data from https://api.sectors.app/v2/q?={query}. Your response should be no longer than 200 words. Return in JSON.*
>
> ”

It is hoped that such prompt engineering instructions will guide the LLM to produce a response that meets the specified criteria above. But beyond a cursory level, prompt engineering alone just isn't enough to address the problems of hallucinations and other enterprise considerations. A prompt like the above is slightly better than no prompt at all, but it does not offer any guarantees that the LLM will actually follow the instructions provided.

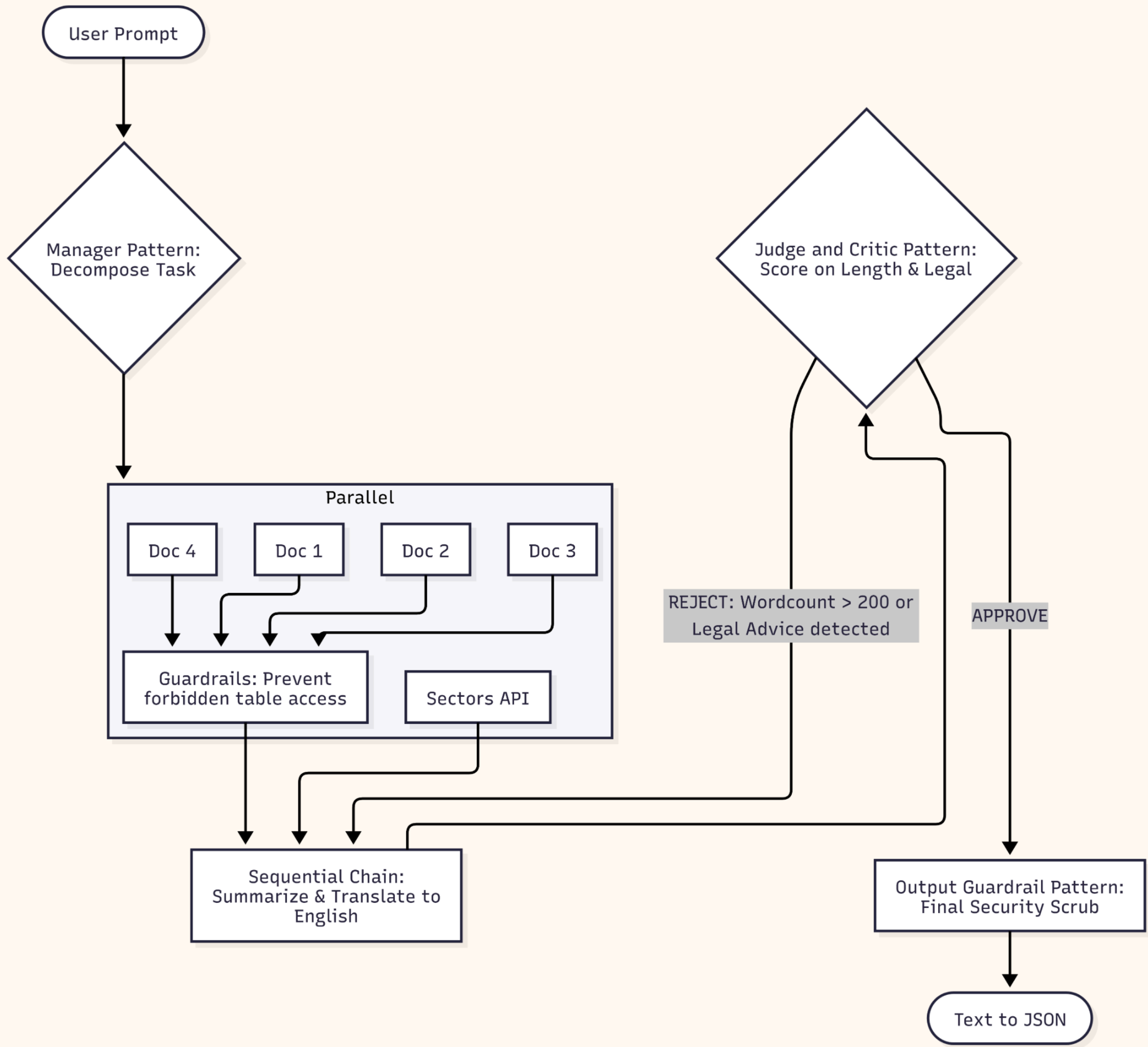
When you think about it – it does make sense that LLMs frequently do ignore parts or all of the prompt engineering instructions. The same pattern recognition capabilities that make LLMs so powerful mean that they are probabilistic in nature and *should, by design, be expected* to generate a string of text that statistically follows from the input prompt.

There is no inherent mechanism within the LLM architecture that enforces compliance with prompt instructions, else we could have just prompted it with "please do not hallucinate" and be done with it.

So a model fundamentally limited by its architecture -- leading to hallucinations -- cannot have those limitations removed from the same model simply by "asking nicely" via prompt engineering. It requires a rethink of the architecture itself, which motivates this book's focus on Agentic AI Patterns.

By the time you have completed this book, you will have been introduced to a toolkit of Agentic Patterns that can be combined to enforce the very same example prompt above -- except on an architectural level, with guarantees that the AI agent will comply with the instructions provided.

Revisit the earlier prompt again. Here is how we would architect the same example prompt using the 6 Agentic Patterns covered in this book.



The architectural diagram above constitutes one such instance of a Blueprint – it combines multiple Agentic Patterns that we will explore in the coming chapters to deliver a robust agentic architecture that mitigates hallucinations and fulfills the enterprise **Considerations** outlined earlier.

Building Agentic Systems

🔖 Chapter Objectives

Understand the building blocks of Agentic AI systems, starting from Single-Agent systems to Multi-Agent systems, and how to implement them using code examples.

Agent Foundations

The singular, most atomic building block of an Agentic AI system is the single **AI Agent**. An Agent comes equipped with its own set of Model, Skills (or Tools), Instructions (think of it as Prompt), and Guardrails. The only relevant pattern at the Single-Agent level is the **Skills and Tools Pattern**, which features the use of tools (such as a database reader, or a PDF viewer, or a web browser) to accomplish a given task. No other patterns are necessary at the level of a Single-Agent system.

As the business case evolves, one might soon find that the list of **Skills and Tools** required to accomplish the task grows – sometimes too large or unwieldy for the single agent to handle. This motivates the need to architect systems with multiple agents collaborating to accomplish a given task.

Sometimes, the decision to pivot from a Single-Agent system to a Multi-Agent system is obvious from the outset as is the case in enterprise use cases – where enterprise Considerations such as Trustworthiness, Scalability, and Governance necessitate the use of multiple agents working in concert to deliver the desired outcome.

In other cases, the decision to move to a Multi-Agent system is more emergent, and less obvious until the system is stress-tested in production.

At Supertype, we encourage the use of multi-agent architectures at the earliest signs of an enterprise Consideration arising. It is infinitely more modular and composable, allowing the engineering team to isolate and address specific Considerations without having to re-architect the entire system from scratch.

Here is an example of a Single-Agent system at work:

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

load_dotenv()

agent = Agent(
    name="Single Agent Assistant",
    model="o3-mini",
    instructions="Translate the documents from Bahasa Indonesia to English
and summarize them in no more than 200 words.",
)

result = Runner.run(agent,
    "Translate and summarize the following documents:",
    inputs={"documents": docs})
```

💡 Tips Prefer LangChain or LlamaIndex over the Agents SDK?

The above code example shows a working code example of a Single-Agent system, implemented using OpenAI's Agents SDK. I also have code examples for LangChain and LlamaIndex on my GitHub repository for readers who prefer those frameworks.

The example leaves out any **Tool Use** (and **Guardrails**) for brevity, and our orchestration uses the `Runner.run()` method to invoke a loop, instructing the Agent to iterate through multiple cycles until an exit condition is triggered. An exit condition might be a tool-use (e.g., "call the database reader tool with these parameters"), or a final answer output (e.g., "the final summary is as follows…"), or when the requirements of a structured output is met, or when a maximum number of iterations is reached, or when an error occurs.

Stepping up to Multi-Agent Systems

As far as complexity is concerned, the Single-Agent system is highly limited in its ability to adequately address the enterprise Considerations, but it is also far easier to implement. As the Considerations get more stringent, or as the scope of the AI Agent's responsibilities expand, a logical step is to add more Skills and Tools to the single agent, allowing the LLM powering the agent to decide on the best tool to use at any given time while remaining constrained within the single agent's instructions.

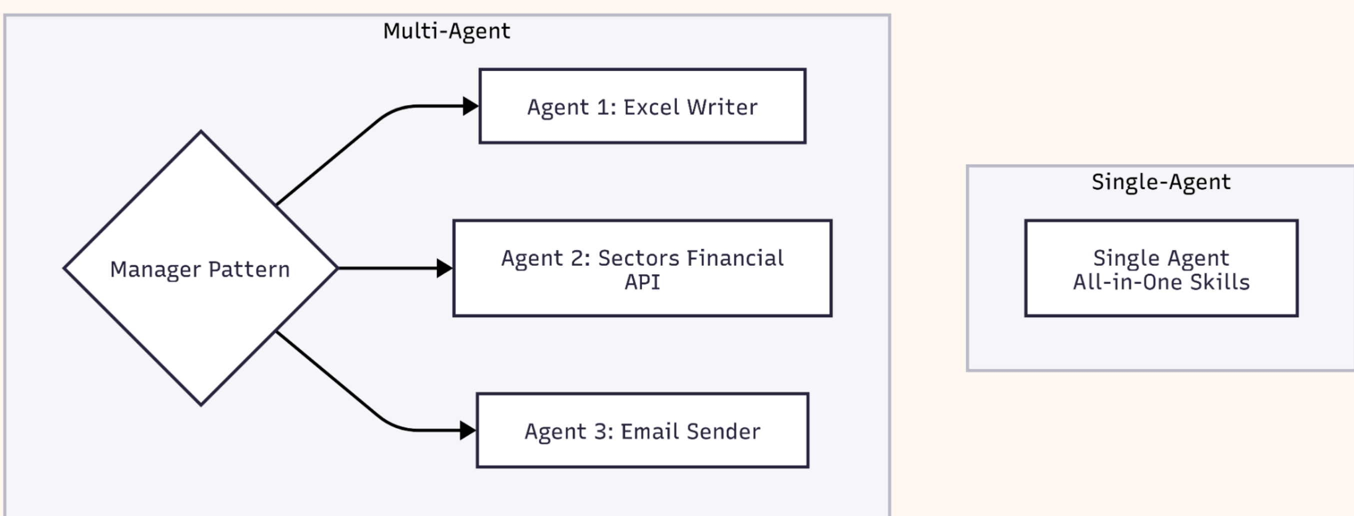
Let's see an example:

```
customer_service_manager = Agent(
    name="Financial Analyst Support",
    instructions="""
    You are a financial analyst at a bank. You are interacting with
{{user_first_name}} who speak {{user_preferred_language}} and whose account
number is {{user_account_number}}. Retrieve the information from the above
account before proceeding to assist {{user_first_name}} with their
account-related queries, in {{user_preferred_language}}.
    """,
    model="o3-mini",
    tools=[...]
)
```

The Agent above uses one LLM model, and is equipped with multiple tools. Instead of multiple agents each specializing in a specific task, or powered by a different LLM model, or operating under different instructions (i.e. answer in English only) and constraints, the Single-Agent here is expected to handle all the tasks by itself.

The Financial Analyst Support agent is being fed with a prompt template that is dynamically populated with user-specific information, and is expected to handle a wide range of queries without the IT department having to provision for a multi-agent system.

We can contrast our Single-Agent system with a Multi-Agent system, where architecturally a Manager agent sits at the top of the orchestration, delegating tasks to specialized Agents below it.



Each specialized Agent might be powered by a different LLM model, have its own set of instructions, and be equipped with its own set of tools. Beyond the obvious benefit in modularity and specialization, the Multi-Agent system also allows for better governance and compliance, as each agent can be designed to adhere to specific regulatory requirements or internal policies. Furthermore, we can have these Sub-Agents operate in parallel (the **Parallelization Pattern**), further improving the system's efficiency and responsiveness.

6 Agentic AI Patterns

🔖 Chapter Objectives

Agentic AI Patterns are reusable architectural patterns that form the building blocks for designing and implementing AI Agents. Learn about these patterns and how they can be combined to create **Blueprints** for various AI use cases.

In the first draft of Agentic Patterns (May 2025), I introduced 6 Agentic AI Patterns (book, accompanying write-up, workshop and code) that have been battle-tested in real-world enterprise deployments. These patterns serve as the foundational building blocks for architecting a **Blueprint**, i.e. an architectural design using consistent terminologies and patterns that are distinctly recognizable in a system.

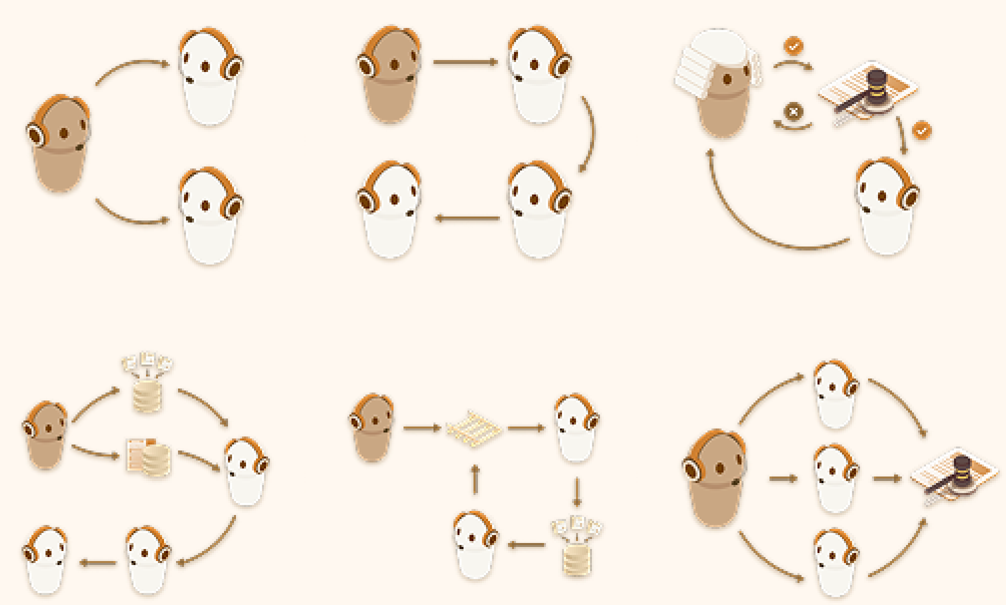
I have since received feedback from readers and practitioners that these patterns have been immensely helpful in providing a common language and framework for structuring conversations around Agentic AI systems – both within engineering teams, and between engineering teams and business stakeholders.

The two updates to the original Agentic Patterns framework include:
(1) a more refined, clearer articulation of each pattern, and (2) a renaming of the previously called "Hand-Off and Delegation Pattern" to "Manager Pattern"; as well as the renaming of the "Tool-Use Pattern" to "Skills and Tools Pattern".

In neither case do these renaming affect the underlying concepts or implementations of the patterns -- they are simply intended to provide clearer, more intuitive names for each pattern, and in the case of the latter, to better align with a term ("Skills") that has rapidly gained traction since being popularized by Antropics' Claude.

As such, here is the updated list of 6 Agentic AI Patterns:

1. The Manager Pattern
2. The Skills and Tools Pattern
3. The Sequential Chain Pattern
4. The Judge and Critic Pattern
5. The Parallelization Pattern
6. The Guardrails Pattern



The Manager Pattern

The Manager Pattern involves the use of a central coordinating agent (the "Manager") that oversees and delegates tasks to multiple specialized sub-agents. The Manager is responsible for breaking down complex tasks into smaller, manageable sub-tasks, assigning these sub-tasks to the appropriate sub-agents based on their expertise or capabilities, and then aggregating the results from these sub-agents to produce a final output.

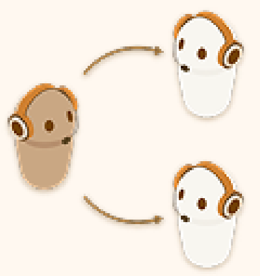
Let's see an example of the Manager Pattern in action:

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

english_agent = Agent(
    name="English Case Specialist",
    instructions="Always respond in English. Be polite and concise."
)

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]
)
```

In the example above, we have a Customer Service Manager agent that delegates customer queries to either a German Case Specialist or an English Case Specialist based on the language of the request. The Manager Pattern allows for efficient handling of diverse tasks by leveraging the strengths of specialized sub-agents, each possibly powered by different LLM models or equipped with different tools.



The Manager Pattern

Involves a central coordinating agent that oversees and delegates tasks to multiple specialized sub-agents, breaking down complex tasks into manageable sub-tasks and aggregating results.

A well designed Manager pattern allows for separation of concerns and modularity, since each Agent is developed and maintained independently (perhaps even by different teams, using different LLM providers). This modularity also facilitates easier updates and improvements to individual agents without affecting the overall system.

A Small Variation: The Hand-Off Pattern

The Hand-Off Pattern is a simplified version of the Manager Pattern, and not a distinct pattern that merits its own category. There is a small but key difference however: In the Manager Pattern, the Manager agent is responsible for decomposing a complex task and delegating sub-tasks to specialized sub-agents – it typically gets combined with the Sequential Chain Pattern to also synthesize the results from the sub-agents before producing a final output.

In the Hand-Off Pattern, as implemented in 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 delegated agent will then take over the conversation and produce the final output, without any further involvement from the original agent.

The Hand-Off Pattern is useful for scenarios where a task can be fully handled by a specialized agent without the need for further decomposition or synthesis, or when the task does not require a central coordinator to manage multiple sub-agents. Just like the Manager Pattern, the Hand-Off Pattern also promotes modularity and can be combined with the Sequential Chain pattern, if desired, such that the delegated agent can itself pass the baton to another specialized agent in a chain of delegation.

The Skills and Tools Pattern

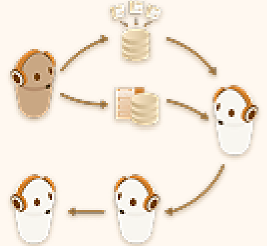
Originally called the "Tool-Use and Function Calling" Pattern, this Pattern is one of the most powerful patterns in agentic AI systems as this provides a key gateway for AI Agents to interact with external systems, databases, APIs, or services that offer them capabilities beyond text generation.

With the right **Tools**, AI Agents can take actions in the world, such as making API calls to retrieve data, perform calculations, surf the web, send emails, write an investment thesis to a Markdown file, or even control IoT devices.

The Agent SDK provides a `function_tool` decorator that makes it easy to define and register tools for an agent to use, but similar libraries like LangChain and LlamaIndex also provide similar capabilities with slightly different syntaxes.

💡 Tips See LangChain and LlamaIndex implementations

While the code example below uses OpenAI's Agent SDK, I have also provided plenty of examples in my LLM in Python repository that demonstrate how to implement the Skills and Tools Pattern using LangChain and LlamaIndex.



The Skills and Tools Pattern

Empowers AI Agents to interact with external systems, databases, APIs, or services through defined tools, enabling them to perform actions beyond text generation.

Creating a Tool for an AI Agent

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 Sectors Financial API, we can build tooling for our AI agents, making them capable of performing complex financial tasks, such as portfolio management, risk assessment, and investment research.

```python
import os
import json
import requests

from dotenv import 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)
```

The utility function above is called `retrieve_from_endpoint`, which takes in a URL string and returns the JSON response from the API endpoint as a Python dictionary.

Let's equip our Single Agent with two Tools, both created using the utility function above.

```
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 find_companies_screener(query: str) -> List[str]:
    url = f"https://api.sectors.app/v2/companies/?q={query}"
    return retrieve_from_endpoint(url)
```

And now onto creating our **Agent** with the aforementioned tools.

```
stock_assistant = Agent(
    name="stock_assistant",
    instructions="Either do stock screening or get company overview based
on user query.",
    tools=[
        get_company_overview,
        find_companies_screener
    ],
    tool_use_behavior="run_llm_again"
)

query = "Screen for IDX companies where Prajogo Pangestu is a major
shareholder."

async def main():
    query = query2
    result = await Runner.run(
        stock_assistant,
        query
    )
    print(f"👧: {query}")
    print(f"🤖: {result.final_output}")
```

When we execute the code above, the AI Agent is able to intelligently decide which tool to use based on the user query. If the user asks for a company overview, it will call the `get_company_overview` tool;

if the user requests a stock screening, it will invoke the `find_companies_screener` tool.

Tool Use Behavior by Agents

Broadly speaking, we are equipping the AI Agent with financial skills, and the Agent's reasoning process can be described as:

1. Understand the user query, and **determine whether a Tool is needed** to fulfill the request
2. If a Tool is needed, **select the appropriate Tool based on the intent** of the user's query
3. When a tool returns a result, **run the LLM again to interpret the result and process it** into a human-readable format (`run_llm_again` as opposed to `stop_on_first_tool` in the `tool_use_behavior` parameter), before finally returning the final output to the user.

Here is the output:

```
➜ python exercise.py
👧: Screen for IDX companies where Prajogo Pangestu is a major shareholder.
🤖: IDX companies where Prajogo Pangestu is a major shareholder:

1. PT Barito Renewables Energy Tbk (BREN.JK)
2. PT Chandra Asri Pacific Tbk (TPIA.JK)
3. Barito Pacific Tbk (BRPT.JK)
4. PT Petrindo Jaya Kreasi Tbk (CUAN.JK)

In all these companies, Prajogo Pangestu is listed among the major shareholders.
```

This is an excellent demonstration of the Skills and Tools Pattern; Instead of making up an answer through linguistic mimicry, our AI Agent is able to leverage an external **Tool** that we created in roughly 15 lines of code, to retrieve fact-based financial data from a trusted source (Sectors Financial Data Platform), and return an accurate, up-to-date answer formatted in Markdown.

We should also note that the Tool is never directly called 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 without any human intervention beyond the initial query. When a tool requires multiple parameters, the Agent relies on the underlying LLM's ability to reason and make a best-effort attempt at filling in the parameters based on the context of the user query.

For example, with the first tool (`get_company_overview`), the Agent is expected to figure out the ticker symbol of the company and the country where said company is listed – both of which are required parameters for the tool to function correctly. Since the raw output of the tool (a JSON string) is not user-friendly, the Agent then invokes the LLM again to interpret the JSON response and format it into a human-readable summary.

This Skills and Tools pattern circumscribes the LLM's tendency to hallucinate, and grounds its responses in verifiable data sources. It also solves the knowledge cutoff problem, since our Tool retrieves data directly from our Knowledge Base that is continuously updated with fresh financial data.

💡 Tips Wiring up a Tool-Use Agent

While you run this example above, please ensure that you have a valid Sectors API key stored in your `.env` file as `SECTORS_API_KEY`.
You can sign up for an account at sectors.app/api. To see a tool-use Agent in action without running it locally or writing any code, you can try Sectors AI Search or Sectors AI Chat.

Content Truncated

You are reading a preview version of the book. The complete version will be released on Supertype's website along with our Blueprints for Enterprise AI workshop series.

Putting it all together: The Blueprint

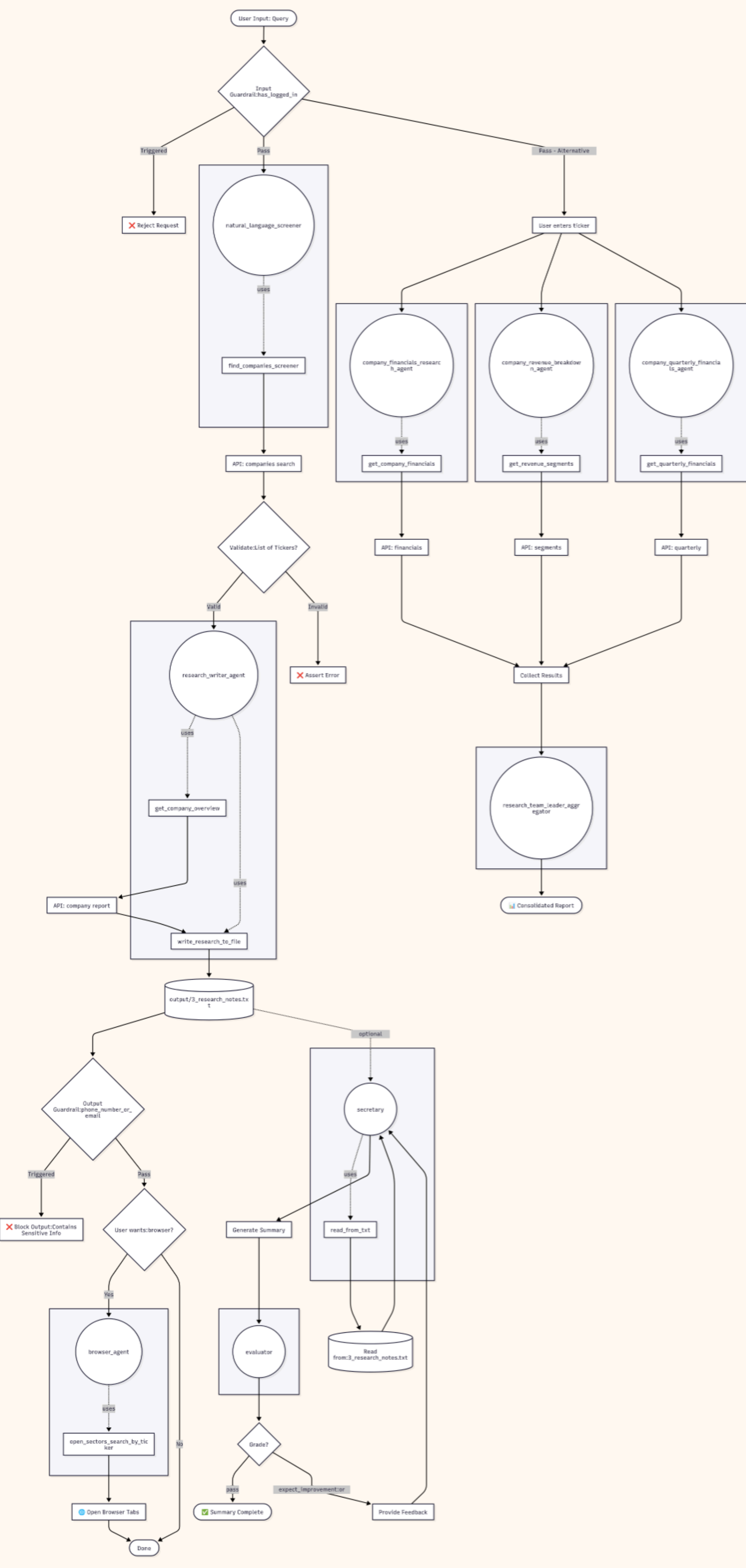
To many readers, the idea of building Agentic AI systems that interact with their company's data, tools, workflows and existing processes seem like a daunting and messy task. The reality is that these Agentic Patterns are already in use today, deployed across tens of organizations that Supertype has had the privilege of working with.

The Agentic patterns discussed in this book are a collection of these past experiences, and we hope that they can serve as an instructional guide for other companies looking to build well-architected Multi-Agent systems of their own. The list of Patterns might not be exhaustive, as the agentic AI space is still evolving rapidly, but the ones presented here lay a solid foundation for any company looking to change the way they build, ship and serve customers in this new era of AI.

Finally, I'd like to thank the team at Supertype for their hard work in building the Sectors Financial AI Suite, without which much of this Book's exercises and practices would not have been possible. Special thanks as well to the many thousands of early adopters at Sectors, whose feedback and suggestions have been immensely valuable.

I also want to thank you, the reader, for taking the time to read this book and for your interest in Agentic AI. Get in touch with us at Supertype if you would like to learn more about our work in this space, and how we can be a development partner as your organization embarks on its own AI journey.

In closing, you will find on the following page a complete Blueprint that ties together all 6 Agentic Patterns discussed throughout this Book. Use it as a reference, a starting point, or a checklist as you design your own Multi-Agent systems for your enterprise.


User Input: Query
Input Guardrail:has_logged_in
Triggered
Pass
Pass - Alternative
❌ Reject Request
natural_language_screener
uses
find_companies_screener
User enters ticker
company_financials_research_agent
company_revenue_breakdown_agent
company_quarterly_financials_agent
uses
get_company_financials
uses
get_revenue_segments
uses
get_quarterly_financials
API: companies search
Validate:List of Tickers?
API: financials
API: segments
API: quarterly
Valid
Invalid
research_writer_agent
❌ Assert Error
Collect Results
uses
get_company_overview
research_team_leader_aggregator
uses
API: company report
📊 Consolidated Report
write_research_to_file
output/3_research_notes.txt
optional
Output Guardrail:phone_number_or_email
secretary
Triggered
Pass
uses
❌ Block Output:Contains Sensitive Info
User wants:browser?
Generate Summary
read_from_txt
Yes
browser_agent
evaluator
Read from:3_research_notes.txt
uses
open_sectors_search_by_ticker
No
Grade?
pass
expect_improvement:or
🌐 Open Browser Tabs
✅ Summary Complete
Provide Feedback
Done

Where to go from here

I am writing this book as part of Supertype's ongoing commitment to disseminate knowledge and best practices in the field of Agentic AI. By the time you are reading this, we should have been working on a follow-up book centered on the **Deliverables** of Enterprise AI.

The Enterprise AI Agentic Stack will be a natural progression from the Blueprint discussed here, as the key themes **shift the focus from the "architectural design" of Multi-Agent systems to the "operationalization and materialization" of these Blueprints into production grade systems** that form the backbone of enterprise AI offerings.

Book	Key Themes	Frameworks
1 Blueprints from Agentic Patterns	The 6 Agentic AI Patterns for orchestrating Multi-Agent Enterprise Systems	📐 6 Agentic Patterns
2 The Enterprise AI Agentic Stack	Building and Managing Agentic Systems using established Blueprints	🌿 LEAVES Deliverables

Other Resources

For readers interested in exploring further, here are some additional resources that complement the content of this book, also written by me and the Supertype team:

- LLM in Python: LLM in Python, featuring more than 40 lessons, long-form explanation videos, and example scripts
- Sectors Financial API: Tutorials and Recipes: 5-Part Series on Generative AI in Finance
- Supertype Publications: All publications and e-books from Supertype
- 6 Agentic Patterns on Sectors Bulletin: An earlier version of the Agentic Patterns (2025), with a heavier emphasis on code examples than conceptual illustrations