Agentic AI Systems

Foundations, Patterns, and Architectures

The MUTTA Approach to Building Production-Grade AI Agent Systems

Alejandro Garcia Polo

Dedicated to

My parents, Cristina and Pedro for their unwavering support and belief in me

and

My partner, Rafaela
for being my constant inspiration

Preface

This book emerged from a simple observation: while the AI community focuses intensely on model capabilities and prompt engineering, systematic architectural patterns for building robust agent systems remain underexplored. Yet it is precisely these patterns—the organizational structures, error correction strategies, and compositional principles—that distinguish prototypes from production systems.

The MUTTA architecture (Manager, Utilities, Tools, and Agents) represents a distillation of best practices from building real-world agentic systems. It provides a systematic approach to structuring agent services that are modular, maintainable, and scalable. More importantly, it offers a framework for reasoning about agent system design that transcends any particular technology or framework.

This book is intended for practitioners: software engineers, ML engineers, researchers, and architects building AI agent systems. It assumes basic familiarity with programming (particularly Python) and large language models, but not deep expertise in either. The focus is practical—every pattern and principle is illustrated with complete working examples.

The field of AI agents is evolving rapidly. By the time you read this, new models will have been released, new frameworks will have emerged, and new capabilities will be available. However, the architectural principles remain stable. Good design principles for managing complexity, ensuring reliability, and maintaining code quality are timeless.

I hope this book serves both as a practical guide for your current projects and as a foundation for adapting to future developments in the field.

— Alejandro Garcia Polo
October 10, 2025

Contents

Chapter 1

Foundations of Agent Architectures

The landscape of artificial intelligence has undergone a profound transformation with the advent of large language models (LLMs) and the agentic systems built upon them. What began as simple prompt-response mechanisms has evolved into sophisticated architectures capable of autonomous reasoning, tool use, and complex multi-step problem solving. This chapter establishes the foundational concepts necessary to understand, design, and implement robust AI agent systems.

Our goal is not merely to build AI systems that work, but to construct architectures that are *simple*, *interchangeable*, and *functional*—systems that behave as reliable services, maintaining consistency across varying model capabilities and adapting gracefully to different problem domains. To achieve this, we must first understand the progression from basic prompts to fully autonomous agents, the fundamental primitives that power these systems, and the architectural patterns that govern multi-agent collaboration.

1.1 Introduction and Agent Levels

1.1.1 Goals: Simple, Interchangeable, and Functional AI Services

Before diving into the technical details of agent architectures, we must establish what we aim to achieve. The primary objective of this book is to present a systematic approach to building AI agent systems that adhere to three core principles:

Simplicity: An effective agent architecture should be understandable and maintainable. Complexity is not a virtue in system design—it is a liability. By decomposing systems into well-defined components with clear responsibilities, we enable developers to reason about behavior, diagnose issues, and extend functionality without unraveling a tangled web of dependencies.

Interchangeability: The rapid pace of progress in AI model development means that today's state-of-the-art model will be tomorrow's baseline. An architecture designed around the idiosyncrasies of a specific model is fragile and short-lived. Instead, we seek patterns that abstract over model capabilities, allowing us to swap underlying LLMs with minimal architectural changes. This principle extends to components within our systems—agents, tools, and utilities should be modular and replaceable.

Functionality: Above all, our systems must work reliably. They should behave as *services*—repeatable functions that accept well-defined inputs and produce predictable outputs. This service-oriented mindset, which we will develop throughout this book, enables us to compose complex behaviors from simple,

tested components and to reason about system correctness at multiple levels of abstraction.

These principles are not merely aesthetic preferences; they are practical necessities for building production-grade AI systems. The architectural patterns we develop in subsequent chapters—particularly the MUTTA (Manager, Utilities, Tools, and Agents) framework—are direct expressions of these principles.

1.1.2 Distinguishing Prompts from Agents

A fundamental distinction underlies all of the architectures we will explore: the difference between a *prompt* and an *agent*. While these terms are often used interchangeably in casual discourse, they represent fundamentally different computational paradigms with distinct capabilities and use cases.

A **prompt** is a single invocation of a language model. It consists of an input (typically text, though modern multimodal models accept other data types) and produces an output (again, typically text or structured data). The prompt is deterministic in structure—you provide input, the model processes it in a single forward pass, and you receive output. There is no iteration, no decision-making about whether to continue processing, and no autonomous tool use.

An **agent**, by contrast, is characterized by autonomy and iterative reasoning. An agent wraps one or more prompts in a control loop, making decisions about what actions to take, when to use tools, and when it has completed its task. The key distinction is *loose determinism*—while we can specify the agent's goals and constraints, we cannot predict the exact sequence of steps it will take to achieve those goals.

Understanding when to use a simple prompt versus when to employ a full agent is crucial for effective system design. Not every problem requires the complexity and computational cost of an agentic solution. A well-designed architecture uses the simplest mechanism sufficient for each task.

The Progression of Complexity: Single Prompt to Agent

The evolution from prompt to agent is not binary but represents a spectrum of increasing capability and complexity. Understanding this progression helps us select the appropriate level of sophistication for each component in our systems.

Level 0: Single Prompt

The simplest form of LLM interaction is the single prompt. We provide context and instructions, invoke the model once, and receive a response. This pattern is appropriate for straightforward tasks with well-defined inputs and outputs:

- Generating a title or summary for a document
- Translating text between languages
- Classifying input into predefined categories
- Extracting structured information from unstructured text

The single prompt is deterministic in structure (though not necessarily in output, given the stochastic nature of language models) and has no mechanism for iteration or tool use.

Level 1: Chained Prompts

The next level of complexity involves chaining multiple prompts in a predetermined sequence. Each prompt in the chain performs a specific transformation, and the output of one becomes the input to the next. This pattern is useful for multi-step transformations where each step is well-defined:

- Pipeline processing: extract entities, then classify them, then generate a report
- Refinement loops: generate draft, critique draft, revise based on critique
- Staged reasoning: break down problem, solve sub-problems, synthesize solution

Chained prompts maintain structural determinism—the sequence of steps is fixed and predetermined. However, the outputs at each stage may vary based on the input and model behavior.

Level 2: Prompts with Tool Calls

A significant capability leap occurs when we enable a single prompt to invoke tools. Here, the model not only generates text but can also produce structured function calls that the system executes on its behalf. The model receives the tool's output and incorporates it into its response.

This level introduces conditional behavior—the model decides *whether* to use a tool and *which* tool to use based on the input and its reasoning. However, the interaction remains a single round: input $\rightarrow$ model decision $\rightarrow$ optional tool call $\rightarrow$ output.

Common use cases include:

- Information retrieval: calling a search API to augment responses
- Computation: invoking a calculator for arithmetic operations
- Data access: querying a database for specific information

Level 3: Multi-Step Tool-Calling Systems

Combining chained prompts with tool-calling capability yields multi-step systems where predetermined stages each have access to tools. This provides both structure (the sequence of stages is fixed) and flexibility (each stage can adapt its tool use to the context).

This pattern is particularly useful for complex workflows with distinct phases:

- Research pipelines: gather sources, extract information, synthesize findings
- Data processing: fetch data, transform it, validate results, store output
- Content creation: research topic, outline structure, write sections, format output

Level 4: Autonomous Agents

The final level in our progression is the true autonomous agent. Unlike the previous levels, an agent does not follow a predetermined sequence of steps. Instead, it operates in a loop, continuously making decisions about what to do next based on its current state, its goals, and the results of previous actions.

An agent can:

- Choose between generating text output and invoking tools
- Decide dynamically which tools to use and in what order
- Determine when it has sufficient information to complete its task
- Iterate and refine its approach based on intermediate results
- Recover from errors and adjust its strategy

This autonomy comes at a cost: agents are less predictable, potentially more expensive (due to multiple model invocations), and require careful design to ensure they remain aligned with user goals. However, for complex, open-ended tasks where the solution path cannot be predetermined, agents are essential.

Core Agent Activities: Text Generation and Tool Execution Primitives

Regardless of the complexity level, all LLM-based systems—from simple prompts to sophisticated multi-agent architectures—are built upon two fundamental primitives:

Primitive 1: Text Generation

The first primitive is the core capability of language models: generating text. Given an input context (which may include system instructions, conversation history, and user input), the model produces a text response. This response may be freeform natural language or structured according to a specified schema (e.g., JSON output following a Pydantic model).

Text generation is the foundation of all model capabilities. Even when a model appears to be "reasoning" or "planning," it is fundamentally generating tokens sequentially, leveraging patterns learned during training to produce coherent and contextually appropriate text.

Primitive 2: Tool Execution

The second primitive is tool execution. Instead of (or in addition to) generating text for the end user, the model generates a structured function call. This call specifies:

- The tool to invoke (selected from a set of available tools)
- The arguments to pass to that tool
- Optionally, the expected return type

The system executes this function call, obtains the result, and provides it back to the model as additional context. This primitive extends the model's capabilities beyond its training data, enabling it to:

- Access real-time information (via APIs and databases)
- Perform precise calculations (via code execution)
- Modify external state (via file operations, API calls, etc.)
- Interact with specialized systems (via custom tool implementations)

Composition and Control Flow

All agent architectures are compositions of these two primitives, orchestrated by control flow logic:

- A *single prompt* uses text generation once
- A *prompt with tool calling* uses both primitives sequentially
- A *chained system* composes multiple instances of these primitives
- An *agent* wraps these primitives in a loop with conditional logic

Understanding this reduction to primitives is crucial for two reasons. First, it demystifies agents—they are not fundamentally different computational entities but rather control structures over familiar operations. Second, it informs our architectural choices—by recognizing that all complexity emerges from composition of two simple primitives, we can design systems that are modular, testable, and maintainable.

This primitive-based view also illuminates the role of agent frameworks. Frameworks like LangChain, AutoGen, and OpenAI's Agents SDK do not add new primitives; they provide abstractions and utilities for common composition patterns. A framework-free agent might explicitly implement a while loop that alternates between model calls and tool execution, while a framework encapsulates this pattern behind a higher-level API. Both approaches use the same primitives.

1.1.3 Defining the Autonomous Agent

Having distinguished agents from simpler prompt-based systems, we now provide a precise definition of what constitutes an autonomous agent in our framework.

An **autonomous agent** is a computational entity that:

1. **Has a goal or objective**: The agent is directed toward completing a task, answering a question, or achieving a specified state.
2. **Makes decisions**: The agent chooses from available actions (text generation, tool use, termination) based on its current context and reasoning.
3. **Operates iteratively**: The agent's behavior is structured as a loop, not a fixed sequence. It continues processing until it determines that its goal has been achieved or that further progress is impossible.
4. **Exhibits loose determinism**: While we define the agent's capabilities and objectives, we do not fully prescribe its behavior. Given the same input, an agent might take different paths to the solution across different invocations.
5. **Uses tools and text generation**: The agent has access to both core primitives and decides when and how to use each.

This definition is operational rather than philosophical. We are not claiming that such agents possess consciousness, true understanding, or general intelligence. Rather, we define agents by their behavioral characteristics and architectural properties.

Autonomy and Loose Determinism in Agent Behavior

The concept of *loose determinism* deserves special attention, as it is both the source of agents' power and their primary challenge from a systems design perspective.

What is Loose Determinism?

In traditional software, we expect deterministic behavior: given the same input, a function produces the same output. This predictability is essential for testing, debugging, and reasoning about system correctness.

Language models, even without agentic scaffolding, are already somewhat non-deterministic due to sampling during generation. However, this non-determinism is typically bounded—the model generates a single response, and while that response may vary across invocations, the structure of the interaction is fixed.

Agents introduce a deeper form of non-determinism. Given the same starting input, an agent might:

- Take different numbers of steps to reach a solution
- Use different tools in different orders
- Explore different reasoning paths
- Reach different conclusions (especially for open-ended tasks)

This variability arises from the agent's decision-making process. At each step, the agent evaluates its current state and decides on its next action. These decisions are influenced by:

- The model's weights and training (deterministic for a given model)
- The sampling process during generation (stochastic)
- The agent's conversation history (path-dependent)
- External tool outputs (potentially non-deterministic)

We call this *loose determinism* because, while the agent's behavior is not random (it is guided by its instructions and reasoning), neither is it fully predictable. The agent has autonomy within the bounds of its capabilities and constraints.

Why Loose Determinism is Valuable

Loose determinism is not a bug to be eliminated but a feature to be harnessed. It provides several advantages:

Adaptability: Different problem instances may require different solution strategies. A loosely deterministic agent can adapt its approach to the specifics of each input, rather than forcing all inputs through a fixed pipeline.

Robustness: If an agent's first approach fails (e.g., a tool call returns an error), it can try alternative strategies. This resilience is difficult to achieve in rigid, predetermined workflows.

Generality: A single agent can handle a wider range of inputs than a fixed pipeline, as it can adjust its behavior to match the input's characteristics.

Managing the Challenges of Loose Determinism

The power of loose determinism comes with challenges:

Testing and Validation: How do we verify that an agent works correctly if its behavior varies across invocations? We must shift from expecting identical outputs to defining acceptable output properties and success criteria.

Debugging: When an agent fails, understanding why requires examining its decision-making process, not just its final output. This necessitates comprehensive logging and observability.

Cost Control: An agent that can take arbitrary numbers of steps might consume excessive compute resources on difficult inputs. We need mechanisms to bound agent behavior (e.g., maximum step counts, timeout constraints).

Correctness Guarantees: For critical applications, loose determinism is problematic. We may need to constrain agent autonomy, add verification steps, or employ error correction mechanisms (topics we address in Chapter 5).

The MUTTA architecture, which we introduce in Chapter 2, provides patterns for managing these challenges while preserving the benefits of agent autonomy.

The Spectrum of Autonomy

Not all agents require the same degree of autonomy. We can design systems with varying levels of flexibility:

Highly Constrained Agents: These agents have limited decision-making freedom. They might choose between a small set of tools or decide when to terminate, but their overall behavior is tightly prescribed. Such agents are appropriate when we need predictability and can anticipate most scenarios.

Moderately Autonomous Agents: These agents have broader freedom to choose tools and strategies but operate within well-defined boundaries (e.g., read-only access to data, limited to specific domains). This is the sweet spot for many production applications.

Highly Autonomous Agents: These agents have extensive freedom to explore solution spaces, use diverse tools, and operate over long time horizons. They are powerful but require careful monitoring and robust error correction. Research systems and specialized applications (e.g., autonomous research assistants) often employ this level of autonomy.

The appropriate level of autonomy depends on your use case, risk tolerance, and the quality of your error correction mechanisms. As a general principle: *grant agents the minimum autonomy necessary to accomplish their tasks effectively.*

1.2 Classification of Multi-Agent Systems

While single agents can accomplish impressive tasks, many complex applications require multiple agents working together. The architecture of multi-agent systems—how agents are connected, how they communicate, and how they coordinate—fundamentally shapes the system's capabilities and characteristics.

We can classify most multi-agent systems into three broad categories based on their coordination patterns: parallel systems, handoff (or expert) systems, and systematic (or in-series) systems. Each category represents different trade-offs between speed, specialization, generality, and reliability.

1.2.1 Parallel Systems (Speed over Specialization)

In a parallel multi-agent system, multiple agents work simultaneously on different aspects of a problem or on different problem instances. The agents may communicate with each other to share information, but they are not dependent on sequential handoffs—each can make progress independently.

Architectural Characteristics

- **Concurrent execution**: Multiple agents operate at the same time, often on separate threads or processes.
- **Shared or partitioned state**: Agents might share access to common data structures or work on separate partitions of a larger dataset.
- **Communication channels**: Agents can exchange messages or updates, though this is optional.
- **Aggregation of results**: A coordinator (which may itself be an agent) collects and synthesizes the outputs from parallel agents.

Advantages

Speed: The primary benefit of parallel systems is throughput. By distributing work across multiple agents, we can complete tasks faster than a single agent working sequentially.

Scalability: Parallel systems can handle increased load by adding more agents, assuming the work can be partitioned effectively.

Coverage: When the goal is to explore a large space or process many items, parallel agents provide better coverage than a single agent with the same time budget.

Trade-offs and Limitations

Limited Specialization: Parallel agents typically perform similar tasks on different inputs. While they may have minor variations in their capabilities, they are not deeply specialized experts in different domains.

Coordination Overhead: Managing parallel execution, preventing conflicts, and aggregating results adds complexity to the system.

Reduced Depth: Parallel systems are best suited for tasks where each agent's work is relatively independent. They are less effective for deeply sequential problems where each step depends on the previous one.

Use Cases

Parallel multi-agent systems excel in scenarios such as:

- **Batch processing**: Analyzing many documents, processing multiple customer requests, or evaluating numerous data records.
- **Distributed search**: Multiple agents exploring different regions of a search space to find solutions faster.
- **Ensemble methods**: Several agents generating solutions independently, then voting or averaging to produce a robust final answer.
- **Load distribution**: Handling high-volume requests by distributing them across a pool of agents.

Example: Parallel Document Analysis

Consider a system that must analyze a large corpus of legal documents to extract specific clauses. A parallel architecture might:

1. Partition the documents into batches
2. Assign each batch to a separate agent
3. Each agent extracts relevant clauses from its documents
4. A coordinator agent aggregates the results and identifies patterns

This approach achieves high throughput by leveraging parallelism, though each agent performs a relatively simple extraction task rather than complex specialized reasoning.

1.2.2 Handoff/Expert Systems (Generality and Long-Term Horizon)

Handoff systems, also called expert systems in multi-agent parlance, organize agents as specialists in different domains or capabilities. When a task requires expertise beyond an agent's scope, it hands off to an appropriate specialist.

Architectural Characteristics

- **Specialized agents**: Each agent is designed for specific types of tasks or domains of knowledge.
- **Handoff mechanism**: Agents have tools that transfer control to other agents, typically passing context and state.
- **Dynamic routing**: The path through the system is not predetermined; agents decide which specialist to invoke based on the current need.
- **Flexible topology**: The handoff graph can be complex, with agents potentially handing off to multiple specialists and even back to previous agents.

Advantages

Generality: A well-designed handoff system can handle a wide variety of tasks by routing each to the appropriate specialist. The system as a whole is general, even though each component is specialized.

Deep Expertise: Each agent can be optimized for its domain, with tailored prompts, specialized tools, and appropriate model selection.

Long-Term Horizon: Complex tasks requiring multiple distinct competencies can be accomplished through a series of handoffs, with each specialist contributing its part.

Modularity: New specialists can be added to expand the system's capabilities without modifying existing agents.

Trade-offs and Limitations

Reduced Speed: Handoffs introduce latency. Each transition between agents requires context transfer and a new model invocation.

Design Complexity: Determining the right set of specialists and designing effective handoff logic is challenging. Poor design leads to confusion about which agent should handle which tasks.

Context Loss: Information can be lost or distorted as it passes between agents, especially if handoff protocols are not carefully designed.

Difficult to Scale: Unlike parallel systems where scaling means adding more identical agents, scaling a handoff system requires identifying which specialists are bottlenecks and carefully adding new capabilities.

Use Cases

Handoff systems are ideal for:

- **Customer service**: Routing inquiries to specialists in billing, technical support, product information, etc.
- **Research workflows**: Handing off between literature review, experimental design, data analysis, and report writing specialists.
- **Software development**: Coordinating between requirements analysis, code generation, testing, and documentation agents.
- **Creative projects**: Transitioning between brainstorming, outlining, drafting, editing, and formatting specialists.

Example: Technical Support System

A technical support handoff system might include:

- A *triage agent* that understands the user's issue and routes to the appropriate specialist
- A *billing specialist* for payment and subscription issues
- A *technical specialist* for product functionality questions
- An *account specialist* for login and security concerns
- An *escalation specialist* for complex cases requiring human intervention

Each specialist has deep knowledge of its domain. The system achieves generality by having specialists for each common issue type, and it handles complex cases through sequential handoffs (e.g., triage → technical → escalation).

1.2.3 Systematic/In-Series Procedures (Reliability and Structure)

Systematic or in-series multi-agent systems organize agents in a predetermined sequence or workflow. Each agent in the series performs a specific step, passes its output to the next agent, and the process continues until completion.

Architectural Characteristics

- **Fixed workflow**: The sequence of agents is predetermined and usually does not vary (though conditional branches may exist).

- **Clear interfaces**: Each agent has a well-defined input and output contract, ensuring smooth data flow through the pipeline.
- **Sequential processing**: Unlike parallel systems, each agent waits for its predecessor to complete before beginning work.
- **State accumulation**: Information flows forward through the pipeline, with each agent potentially adding to or refining the accumulated state.

Advantages

Reliability: The predictable structure of systematic workflows makes them easier to test, debug, and verify. You can define expected states at each stage and validate that the system meets those expectations.

Clarity: The system's behavior is transparent. Stakeholders can understand what the system does by examining the sequence of steps.

Error Isolation: When something goes wrong, the structured nature of the workflow helps identify which stage failed and why.

Optimization: Each stage can be independently optimized for its specific task, including choice of model, prompts, and tools.

Trade-offs and Limitations

Reduced Generality: Systematic workflows are designed for specific types of inputs. They handle expected cases well but struggle with inputs that don't fit the assumed pattern.

Speed: Sequential processing means latency accumulates. The total time is the sum of all stage durations, without the benefits of parallelization.

Rigidity: Adapting to new requirements often means redesigning the entire workflow rather than adding a new specialist.

Design Effort: Creating an effective systematic workflow requires careful upfront analysis to identify the right stages and interfaces.

Use Cases

Systematic systems are best suited for:

- **Data processing pipelines**: ETL (Extract, Transform, Load) workflows where data passes through defined transformation stages.
- **Content generation workflows**: Research → outline → draft → edit → format → publish.
- **Compliance and approval processes**: Submit → review → validate → approve → execute.
- **Quality assurance**: Generate → test → fix → verify → deploy.

Example: Academic Paper Writing System

An academic paper writing workflow might include:

1. **Research Agent**: Gathers relevant literature and extracts key findings
2. **Synthesis Agent**: Identifies themes and connections across sources
3. **Outlining Agent**: Creates a structured outline for the paper

4. **Drafting Agent**: Writes each section based on the outline and research
5. **Citation Agent**: Ensures all claims are properly cited
6. **Editing Agent**: Improves clarity, style, and flow
7. **Formatting Agent**: Applies journal-specific formatting requirements

This systematic approach ensures that each critical step is completed in the right order, with appropriate attention to detail at each stage. While less flexible than a handoff system, it provides reliability and consistency for a well-defined task.

Hybrid Architectures

In practice, sophisticated systems often combine elements from multiple categories:

- A systematic workflow where one stage employs parallel processing (e.g., the research stage runs multiple search agents in parallel)
- A handoff system that uses systematic workflows within each specialist
- A parallel system that coordinates through a series of synchronization points

Understanding these fundamental patterns allows you to design hybrid architectures that leverage the strengths of each approach for different parts of your system.

1.3 Frameworks and Philosophies

The practical work of building agent systems requires choosing tools and frameworks. However, beyond the specific APIs and libraries lies a more fundamental choice: the philosophy that guides how we think about abstraction, composition, and control in agentic systems.

1.3.1 The Philosophy of Primitives vs. Abstraction

The AI agent framework landscape presents developers with a spectrum of choices, from batteries-included frameworks that provide high-level abstractions for common patterns to minimal SDKs that expose fundamental primitives with little scaffolding. Understanding this spectrum is essential for making informed architectural decisions.

The Case for Rich Abstractions

Frameworks like Microsoft's AutoGen, LangChain, and CrewAI sit toward the high-abstraction end of the spectrum. They provide:

- **Pre-built patterns**: Common agent architectures (e.g., ReAct, multi-agent debates) are implemented and ready to use.
- **Extensive utilities**: Tools for memory management, conversation history, result caching, and more come built-in.

- **Opinionated structures**: The framework guides you toward specific ways of organizing agents, tools, and workflows.
- **Rich ecosystems**: Large communities contribute integrations, examples, and extensions.

These frameworks lower the barrier to entry. A developer can create a functioning multi-agent system in dozens of lines of code, leveraging battle-tested abstractions rather than reinventing common patterns.

However, rich abstractions come with costs:

- **Learning overhead**: Understanding the framework's concepts, APIs, and idioms requires significant investment.
- **Reduced transparency**: High-level abstractions obscure what's actually happening, making debugging and optimization more difficult.
- **Lock-in**: Committing to a framework's patterns and structures makes it harder to migrate to alternatives or to drop down to lower-level control when needed.
- **Inflexibility**: When your needs diverge from the framework's assumptions, fighting against the framework's opinions becomes costly.

The Case for Primitive-Based Approaches

At the opposite end of the spectrum are minimal SDKs that provide just enough structure to handle the core primitives—text generation and tool execution—while leaving higher-level patterns to the developer. Examples include OpenAI's Agents SDK and smolagents, a lightweight library for building agents.

These primitive-based approaches offer:

- **Transparency**: You can see and understand exactly what your system is doing at each step.
- **Flexibility**: Without strong framework opinions, you can architect your system to fit your specific needs.
- **Simplicity**: Fewer abstractions mean less conceptual overhead and a shallower learning curve for the core SDK (though you'll build your own patterns on top).
- **Portability**: Code built on primitives is easier to adapt to new frameworks or to run framework-free.

The trade-off is that you must implement common patterns yourself. A while loop for agent execution, structured memory management, error handling—these become your responsibility rather than being provided by the framework.

The Middle Ground

LangGraph, a relatively recent addition to the ecosystem, attempts to occupy a middle ground. It provides a graph-based abstraction for agent workflows (nodes and edges representing steps and transitions) while remaining relatively close to the primitives. This approach offers structure without excessive opinion.

Our Recommendation: Start with Primitives

For the purposes of this book, and for production system development in general, we advocate for *starting with primitives*. Specifically, we will use OpenAI's Agents SDK as our primary framework because:

1. **It teaches fundamental concepts**: By working close to the primitives, you develop a deep understanding of how agents actually work, which makes you a better architect regardless of what tools you eventually use.

2. **It enforces good design**: Without pre-built complex abstractions, you must thoughtfully design your system's structure. This often leads to cleaner, more maintainable architectures.

3. **It adapts to the MUTTA pattern**: The primitive-based approach aligns naturally with the MUTTA (Manager, Utilities, Tools, and Agents) architecture we develop in Chapter 2, which emphasizes clear separation of concerns.

4. **It remains stable**: Minimal SDKs have smaller API surfaces and change less frequently than feature-rich frameworks, reducing maintenance burden.

This is not to say that higher-level frameworks are wrong or inferior. For rapid prototyping, for teams with limited ML engineering expertise, or for applications that fit neatly into a framework's paradigm, batteries-included frameworks can be excellent choices. However, for building robust, production-grade systems that you will maintain and evolve over time, we believe the primitive-based approach provides a stronger foundation.

1.3.2 Selecting the SDK: Using OpenAI Agents for Flexibility and Efficiency

Having established our preference for primitive-based frameworks, we now explain why OpenAI's Agents SDK specifically serves as the foundation for examples throughout this book.

Minimalism and Clarity

The OpenAI Agents SDK provides exactly what you need to build agents, and little more:

- An `Agent` class that encapsulates an LLM with system instructions
- A mechanism for defining and registering tools
- Support for structured outputs via JSON schemas or Pydantic models
- Handoff capabilities for multi-agent systems
- A simple execution loop that handles tool calling and response generation

This minimalism is a feature, not a limitation. It keeps the conceptual surface area small, making the SDK easy to learn and understand completely.

Alignment with MUTTA Principles

The SDK's design aligns naturally with the MUTTA architecture:

- Agents are defined declaratively in separate modules (fitting the MUTTA agent directory pattern)
- Tools are regular Python functions with decorators (fitting the tools.py pattern)

- Orchestration logic remains in your code, not hidden in framework magic (fitting the manager.py pattern)
- System instructions and model selection are explicit and visible (supporting maintainability)

Model Flexibility

While the SDK is developed by OpenAI, it is designed to work with both OpenAI's models as well as any other model—either local or deployed—that you have access to.

This means that as long as the models you are targeting satisfy similar needs that the framework requires (such as the ability to call tools, though this is not strictly required, or the ability to produce structured outputs), you will be able to use this SDK with any model you need. The SDK is not tied to OpenAI simply because of its name.

The integration to communicate with any model can be done manually with the SDK, but there is a helper library called LiteLLM (`litellm`) that simplifies this process. By installing the SDK with the LiteLLM dependency (`pip install openai-agents[litellm]`), you can connect to many providers and local models even more easily, without any issues.

Production Readiness

Despite its simplicity, the OpenAI Agents SDK is production-ready:

- It handles error cases gracefully
- It supports async execution for high-concurrency scenarios
- It integrates with standard Python tooling (type hints, testing frameworks, etc.)
- It provides sufficient logging and observability hooks for debugging

Framework-Agnostic Principles

While our code examples use the OpenAI Agents SDK, the architectural principles and patterns we develop are framework-agnostic. If you prefer a different SDK or choose to build framework-free agents, the MUTTA architecture, error correction strategies, and reusable patterns from Chapters 2 through 5 apply equally well.

Think of the SDK as a convenience layer over the primitives, not as a fundamental requirement. The essential concepts are:

- Agents encapsulate LLMs with instructions and tools
- Managers orchestrate agent interactions
- Tools extend agent capabilities
- Utilities provide deterministic helper functions

These concepts exist independently of any particular framework.

A Note on Framework Evolution

The AI tooling landscape evolves rapidly. By the time you read this book, new frameworks may have emerged, and existing ones may have evolved significantly.

This is precisely why we emphasize primitives and architecture over framework-specific details.

If you deeply understand:

- How language models generate text and call tools
- How to structure systems for maintainability and reliability
- How to manage errors and ensure quality in agentic workflows

...then you can adapt to any framework or build without one. The frameworks are tools to make implementation easier; the principles are what make your systems robust.

Getting Started

To work with the examples in this book, you'll need:

1. An LLM that you can connect the SDK with (we recommend using an OpenAI account and API key for testing, as it is fairly reliable)
2. Python 3.9 or later
3. The OpenAI Agents library: `pip install openai-agents`
4. *Optional*: If working with non-OpenAI models: `pip install openai-agents[litellm]`

The basic structure of an agent in this SDK is:

```python
from openai import Agent

# Define system instructions
instructions = """You are a helpful assistant that..."""

# Create the agent
agent = Agent(
    name="example_agent",
    model="gpt-5",
    instructions=instructions,
    tools=[...],  # Optional tools
)

# Run the agent
response = agent.run("User input here")
```

This simplicity is deceptive—from this foundation, we will build sophisticated multi-agent systems with error correction, quality assurance, and production-grade reliability.

With these foundations in place, we are ready to explore the core architectural pattern that will structure our approach to building production agent systems: the MUTTA (Manager, Utilities, Tools, and Agents) architecture, which we develop in Chapter 2.

The journey from understanding what agents are to building reliable, maintainable agent systems requires more than just technical knowledge—it requires

a systematic approach to architecture. In the next chapter, we will see how the service-oriented mindset and the MUTTA pattern provide that systematic approach.

Chapter 2

Designing the Modular AI Service Architecture

In Chapter 1, we explored the progression from simple prompts to autonomous agents and examined patterns for multi-agent coordination. We established that effective agent systems should be simple, interchangeable, and functional. However, understanding these principles conceptually is only the beginning. The challenge lies in translating them into concrete architectural patterns that guide day-to-day development decisions.

This chapter introduces the **MUTTA** architecture—Manager, Utilities, Tools, and Agents—a systematic approach to structuring AI agent systems as modular, composable services. MUTTA is not a framework or library; it is an architectural pattern, a set of organizing principles that can be implemented with any agent SDK or even with no framework at all.

The MUTTA pattern emerged from a simple observation: the most maintainable and reliable agent systems separate *orchestration logic* from *agent behavior* from *deterministic utilities*. By enforcing this separation through a clear file structure and well-defined responsibilities, we create systems that are easier to understand, test, debug, and evolve.

We begin by establishing the service-oriented mindset that underlies MUTTA, then explore each component of the architecture in detail, and conclude with complete working examples that demonstrate the pattern in practice.

2.1 Modeling Complex Systems as Services

Before diving into the specifics of the MUTTA architecture, we must establish a fundamental shift in perspective: we should think of agent systems not as monolithic applications but as *services*—modular, reusable components with well-defined interfaces.

2.1.1 The Service Definition: Input, Output, and Repeatability

In software architecture, a **service** is a self-contained unit of functionality that:

1. **Accepts well-defined inputs**: The service specifies what data it requires and in what format.

2. **Produces well-defined outputs**: The service guarantees what it will return upon successful completion.

3. **Maintains encapsulation**: The service's internal implementation is hidden from consumers; only its interface is visible.

4. **Exhibits repeatability**: Given the same input, the service should produce consistent results (within acceptable bounds of variation).

5. **Handles errors gracefully**: The service defines failure modes and communicates them clearly to consumers.

This service-oriented perspective is common in traditional software engineering, particularly in microservices architectures and API design. However, it is less prevalent in AI system development, where the focus often shifts to model capabilities and prompt engineering rather than architectural discipline.

Why Services Matter for Agent Systems

Applying the service pattern to agent systems provides several critical benefits:

Composability: Services can be composed into larger systems. A complex workflow might involve multiple services, each handling a distinct responsibility. This composition is only possible if each service has a clear, stable interface.

Testability: A service with defined inputs and outputs can be tested in isolation. You can verify its behavior across a range of inputs without needing to understand or mock the entire system.

Reusability: A well-designed service can be used in multiple contexts. For example, a document summarization service might be employed in a research pipeline, a content management system, and a customer support tool.

Maintainability: When a service encapsulates its complexity behind a clean interface, you can modify its internal implementation without affecting consumers. This is essential as models evolve, better prompts are discovered, or optimization opportunities arise.

Scalability: Services with clear boundaries can be deployed independently, scaled according to their specific resource needs, and even distributed across multiple systems.

Translating Agent Systems into Services

To think of an agent system as a service, we must define:

Input Schema: What information does the system need to begin processing? This might be structured data (e.g., a Pydantic model representing a user query and context) or unstructured text, but it should be explicitly defined.

Output Schema: What will the system return? For agent systems, this is often structured data—not just freeform text. Using JSON schemas or Pydantic models for outputs makes services more composable and easier to consume programmatically.

Service Contract: What guarantees does the service make? For example: "Given a valid product description, this service returns a list of relevant marketing keywords, or raises a ValidationError if the description is malformed."

Side Effects: Does the service modify external state (e.g., writing to a database, sending emails)? If so, these side effects should be documented as part of the service contract.

Example: Document Analysis Service

Consider a service that analyzes legal documents to extract key clauses:

```python
from pydantic import BaseModel
from typing import List

class DocumentInput(BaseModel):
    document_text: str
    clause_types: List[str]  # e.g., ["confidentiality", "termination"]

class ExtractedClause(BaseModel):
    clause_type: str
    text: str
    page_number: int
    confidence: float

class DocumentOutput(BaseModel):
    clauses: List[ExtractedClause]
    summary: str

def analyze_document(input: DocumentInput) -> DocumentOutput:
    """
    Service that extracts specified clauses from legal documents.

    Args:
        input: Document text and desired clause types

    Returns:
        Extracted clauses with metadata and a summary

    Raises:
        ValueError: If document_text is empty or clause_types is invalid
    """
    # Implementation using MUTTA architecture (detailed later)
    ...
```

This service has a clear contract: provide a document and clause types, receive structured results. The service can be tested by providing sample inputs and validating outputs. It can be composed with other services (e.g., a document retrieval service upstream, a report generation service downstream). And it can be implemented using agents internally without exposing that complexity to consumers.

Services vs. Functions

You might notice that our service definition closely resembles a function. This is intentional—in Python, a service is often implemented as a function (or a callable class). The distinction is philosophical rather than syntactic:

A *function* is any callable unit of code. A *service* is a function designed with the principles of clear interfaces, encapsulation, and repeatability. Not all functions are services (e.g., a utility function that formats a string), but all services can be exposed as functions.

In the MUTTA architecture, the service is typically implemented as a class with a callable method, providing state management and easier testing through dependency injection.

2.1.2 Service Depth Constraints (The Rule of 20 and Error Correction Requirements)

While the service pattern provides organizational clarity, agent systems introduce a unique challenge: they can theoretically run indefinitely. An agent in a loop might take arbitrarily many steps to complete its task, leading to unbounded latency, cost, and unpredictability.

To maintain the service property of repeatability and reasonable resource consumption, we introduce the concept of **service depth** and establish practical constraints on it.

Defining Service Depth

The *depth* of a service is the maximum number of sequential agent invocations (not parallel invocations) required to process an input. A single unit of depth is defined as:

One depth step = Calling an agent, where that agent invocation contains *at most 10 LLM turns* (individual calls to the underlying language model).

Key clarifications:

- **Utility function calls do not count toward depth**—these are deterministic operations that don't contribute to error accumulation
- **Agent LLM turns are grouped**: If an agent makes 10 or fewer LLM calls during its invocation, that counts as 1 depth step. If it makes 11 calls, that counts as 2 depth steps (ceiling division by 10)
- **Transitions between agents add depth**: Moving from one agent to another sequential agent adds 1 to the depth count

For example, consider a service with the following flow:

1. Manager calls utility to preprocess input (depth = 0, utilities don't count)
2. Manager invokes Agent A, which makes 8 LLM turns (depth = 1, since $\lceil 8/10 \rceil = 1$)
3. Manager calls another utility for data transformation (depth still = 1)
4. Manager invokes Agent B, which makes 15 LLM turns (depth = $1 + \lceil 15/10 \rceil = 1 + 2 = 3$)
5. Transition to Agent C (adds 1): depth = $3 + 1 = 4$
6. Agent C makes 5 LLM turns (depth = $4 + \lceil 5/10 \rceil = 4 + 1 = 5$)

This service has a depth of 5. Note that if Agent B and Agent C ran in parallel (no sequential dependency), we would take the maximum of their depths, not the sum.

The Rule of 20

Through empirical observation across a wide range of agent systems, a practical guideline emerges: *services should maintain a depth of no more than 20 steps before requiring error correction.*

This is not an absolute law but a heuristic based on error propagation characteristics. As we will explore in detail in Chapter 3, errors in agent systems accumulate across sequential steps. The total error of a service can be bounded by:

$$E_{\text{total}} \leq n \times \epsilon_{\max}$$

where n is the number of depth steps and $\epsilon_{\max}$ is the maximum error of any single depth step.

Even with well-designed agents and high-quality models, each depth step introduces some probability of error—whether from hallucinations, misunderstanding context, or incorrect tool use. As the number of steps grows, the cumulative error probability increases.

At depth 20, assuming a modest per-step error rate of 5%, the probability of at least one error occurring somewhere in the service approaches:

$$P(\text{at least one error}) = 1 - (1 - 0.05)^{20} \approx 0.64$$

This empirical observation suggests that approaching 20 steps in depth correlates with increased error rates, and beyond 20 steps, error correction becomes not just beneficial but nearly essential for maintaining service reliability.

Strategies for Managing Depth

When designing a service that would naturally exceed depth 20, you have several options:

Decompose into Multiple Services: Break the complex service into smaller, independent services that each respect the depth limit. These services can be composed by a higher-level orchestrator.

Introduce Error Correction: Incorporate error checking and correction mechanisms at strategic points in your service. We explore these patterns in detail in Chapter 5, including lazy verification agents and overseer patterns.

Reduce Steps per Agent: If agents are taking many turns, consider providing them with better tools, clearer instructions, or more capable models to reduce the number of iterations needed.

Parallelize Where Possible: Sequential steps accumulate depth; parallel steps do not. If portions of your workflow are independent, execute them concurrently.

Measuring Depth in Practice

To monitor service depth in production systems, instrument your code to track:

- The number of agent turns per request
- The number of handoffs between agents
- The number of utility function calls
- The total depth as a sum of the above

This telemetry helps identify services that are approaching or exceeding the depth limit and may benefit from refactoring or error correction.

Depth vs. Latency

It's important to distinguish depth from latency. A service might have low depth but high latency if each step is slow (e.g., using large models, processing substantial data). Conversely, a service might have high depth but acceptable latency if steps are fast.

Depth is primarily a concern for *correctness* (error accumulation) rather than performance. However, the two are related—lower depth generally correlates with lower latency and cost.

Example: Respecting Depth Constraints

Suppose we're building a service that analyzes academic papers. The naive approach might:

1. Extract text from PDF using utility function (depth = 0, utilities don't count)
2. Identify sections with Agent A making 8 LLM turns (depth = $\lceil 8/10 \rceil = 1$)
3. Extract key claims: Agent B makes 25 LLM turns total (depth = $1 + \lceil 25/10 \rceil = 1 + 3 = 4$)
4. Transition to citation agent (depth = $4 + 1 = 5$)
5. Find citations: Agent C makes 18 LLM turns (depth = $5 + \lceil 18/10 \rceil = 5 + 2 = 7$)
6. Transition to synthesis agent (depth = $7 + 1 = 8$)
7. Synthesize findings: Agent D makes 35 LLM turns (depth = $8 + \lceil 35/10 \rceil = 8 + 4 = 12$)

Total depth: 12 steps, which is acceptable and within our guideline.

However, if we added more processing steps that pushed us beyond depth 20, we might refactor as two services:

Service 1: Extraction (Depth 8)

1. Identify sections (Agent A: 8 turns, depth = 1)
2. Extract claims (Agent B: 25 turns, depth = $1 + 3 = 4$)
3. Transition + preliminary verification (Agent C: 15 turns, depth = $4 + 1 + 2 = 7$)

Service 2: Analysis (Depth 6)

1. Verify citations (Agent D: 18 turns, depth = 2)
2. Transition + synthesize findings (Agent E: 20 turns, depth = $2 + 1 + 2 = 5$)

Each service now fits comfortably within the depth limit. A higher-level workflow orchestrates both services, with explicit error handling at the service boundaries.

This decomposition also improves reusability—Service 1 might be useful for other tasks that need claim extraction, and Service 2 might work with claims from sources other than academic papers.

2.2 The Manager, Utilities, Tools, and Agents (MUTTA) Structure

Having established the service-oriented mindset and depth constraints, we now introduce the MUTTA architecture: a file organization and responsibility pattern that embodies these principles.

MUTTA is an acronym for the four core components of an agent service:

- **M**anager: Orchestrates the service logic
- **U**tilities: Provides deterministic helper functions
- **T**ools: Implements functions for agent use
- **T**ools: (Yes, it's mentioned twice—a quirk of the acronym, but both "T"s refer to the same concept)
- **A**gents: Defines autonomous agent behaviors

The structure is intentionally simple and maps directly to a file organization that makes the architecture self-documenting.

2.2.1 Service File Structure Overview

A MUTTA service is organized as follows:

```
service_name/
|-- manager.py            # Orchestration logic (required)
|-- agents/               # Agent definitions (required)
|   |-- agent_a.py
|   |-- agent_b.py
|   `-- ...
|-- tools.py              # Agent-accessible tools (optional)
|-- utilities.py          # Manager helper functions (optional)
`-- __init__.py           # Service interface
```

This structure is not arbitrary—it enforces separation of concerns:

manager.py: Contains *only* the orchestration logic. This file should be highly readable, almost like pseudocode. When you open `manager.py`, you should immediately understand the service's workflow.

agents/: Each agent gets its own file. This enforces modularity and makes it easy to locate and modify specific agent behaviors. Agent files are declarative—they specify what the agent is, not what the service does.

tools.py: Functions that agents can call. These extend agent capabilities but are not called directly by the manager.

utilities.py: Functions that the manager calls. These are deterministic utilities for data transformation, validation, or other non-agentic operations.

The separation between `tools.py` and `utilities.py` is crucial: it clarifies which functions are part of the autonomous agent behavior (tools) and which are part of the deterministic orchestration logic (utilities).

Example Structure: Document Summarization Service

```
document_summarizer/
|-- manager.py
|-- agents/
|   |-- extraction_agent.py       # Extracts key points
|   |-- synthesis_agent.py        # Synthesizes summary
|   `-- formatting_agent.py       # Formats output
|-- tools.py                       # Tools like web_search, fetch_references
|-- utilities.py                   # Utilities like validate_input, chunk_text
`-- __init__.py
```

Minimal Requirements

At minimum, a MUTTA service requires:

- `manager.py` (the orchestrator)
- `agents/` directory with at least one agent file

Both `tools.py` and `utilities.py` are optional, though most non-trivial services will use at least utilities for input validation and preprocessing.

Benefits of This Structure

Discoverability: New developers (or future you) can quickly understand the service by examining the file structure. The roles of different components are immediately clear.

Modularity: Each component can be developed, tested, and modified independently. Changes to an agent don't affect the manager logic, and vice versa.

Reusability: Agents defined in the `agents/` directory can be imported and used in other services. Utilities can be shared across services.

Testability: Each file contains a cohesive unit of functionality that can be tested in isolation.

Maintainability: The enforced separation prevents the common anti-pattern where orchestration logic, agent prompts, tools, and utilities become intertwined in a single monolithic file.

2.2.2 The Manager (`manager.py`): Defining Architectural Flow

The manager is the heart of the MUTTA architecture. It defines *what* happens and *when*, without concerning itself with *how* agents accomplish their tasks.

Manager Responsibilities and Class Requirements

The manager has a focused set of responsibilities:

1. **Define the service interface**: The manager exposes the primary entry point for the service, typically as a callable method.
2. **Orchestrate the workflow**: The manager calls agents, utilities, and handles data flow between steps.
3. **Handle errors**: The manager implements error handling logic, determining how to respond to failures in agents or utilities.
4. **Manage state**: If the service maintains state across steps, the manager coordinates this state.
5. **Return results**: The manager packages the final output according to the service contract.

What the manager does *not* do:

- Define agent behaviors (those belong in `agents/`)
- Implement tool logic (that belongs in `tools.py`)
- Perform complex data transformations (delegate to `utilities.py`)

Manager as a Class

The manager should almost always be implemented as a class, not a standalone function. This provides several advantages:

State Management: The manager can maintain instance variables for configuration, shared resources, or intermediate results.

Dependency Injection: External dependencies (databases, APIs, configuration) can be injected via the constructor, making testing easier.

Reusability: Multiple instances of the manager can run concurrently with different configurations.

Lifecycle Management: The class pattern supports initialization and cleanup logic.

A typical manager class structure:

```
from pydantic import BaseModel
from typing import Optional

class ServiceInput(BaseModel):
    # Input schema definition
    ...

class ServiceOutput(BaseModel):
    # Output schema definition
    ...

class ServiceManager:
    """
    Orchestrates the document analysis service.
    """

```

```python
    def __init__(self, config: Optional[dict] = None):
        """
        Initialize the manager with optional configuration.

        Args:
            config: Service configuration parameters
        """
        self.config = config or {}
        # Initialize any shared resources

    def run(self, input: ServiceInput) -> ServiceOutput:
        """
        Main service entry point.

        Args:
            input: Validated input data

        Returns:
            Service results

        Raises:
            ValidationError: If input is invalid
            ServiceError: If processing fails
        """
        # Step 1: Preprocessing
        preprocessed = self._preprocess(input)

        # Step 2: Agent processing
        agent_result = self._run_agent_a(preprocessed)

        # Step 3: Post-processing
        output = self._postprocess(agent_result)

        return output

    def _preprocess(self, input: ServiceInput):
        """Utility function for input preprocessing."""
        # Calls to utilities.py functions
        ...

    def _run_agent_a(self, data):
        """Invoke Agent A with appropriate context."""
        # Import and run agent
        from .agents.agent_a import agent_a
        result = agent_a.run(data)
        return result

    def _postprocess(self, agent_output):
        """Transform agent output into service output format."""
        ...
```

Readability as a Primary Goal

The manager file should be the most readable file in your service. Someone unfamiliar with the implementation details should be able to open `manager.py`

and quickly understand the service's workflow.

To achieve this readability:

- Use descriptive method names (`_extract_claims`, not `_process_step_2`)
- Keep the main `run` method concise
- Extract complex logic into private methods
- Add comments explaining *why* steps occur, not *what* they do (the code should show what)
- Use type hints extensively

A well-written manager reads almost like a specification document:

```python
def run(self, input: ResearchQuery) -> ResearchReport:
    """Execute the research pipeline."""
    # Validate that we have sufficient information to proceed
    validated_input = validate_research_query(input)

    # Gather sources from multiple databases
    sources = self._gather_sources(validated_input)

    # Extract relevant information using our extraction agent
    extracted_data = self._extract_information(sources)

    # Synthesize findings into a coherent narrative
    synthesis = self._synthesize_findings(extracted_data)

    # Format according to output requirements
    report = self._format_report(synthesis)

    return report
```

This manager clearly shows the pipeline: validate → gather → extract → synthesize → format. The implementation details are delegated to helper methods and agents.

Constraint: Managers Must Not Dynamically Create Agents

A critical constraint in the MUTTA architecture is that **managers must not dynamically create agents**. Agents should be defined statically in the `agents/` directory and imported by the manager, not constructed at runtime based on dynamic conditions.

The Anti-Pattern

Consider this problematic code:

```python
## BAD: Dynamically creating agents in the manager
def run(self, input: ServiceInput) -> ServiceOutput:
    # Determine which agent to create based on input
    if input.task_type == "summarization":
        agent = Agent(
            name="summarizer",
            instructions="Summarize the following...",
            model="gpt-5"
        )
    elif input.task_type == "extraction":
        agent = Agent(
            name="extractor",
            instructions="Extract key points from...",
            model="gpt-5"
        )

    result = agent.run(input.text)
    return result
```

This violates the MUTTA principle because:

- Agent definitions are now embedded in orchestration logic
- The agents cannot be tested independently
- The manager file becomes cluttered with prompt text and configuration
- Reusing these agents in other contexts requires code duplication
- Understanding agent behavior requires reading the manager code

The Correct Pattern
Instead, define agents statically:

```python
## agents/summarization_agent.py
from openai import Agent

instructions = """You are an expert at creating concise, accurate
summaries. Given a text, extract the key points and present them
in a clear, structured format."""

summarization_agent = Agent(
    name="summarization_agent",
    model="gpt-5",
    instructions=instructions
)
```

```python
## agents/extraction_agent.py
from openai import Agent

instructions = """You are an expert at information extraction.
Given a text and a list of target information types, extract
all relevant instances."""

extraction_agent = Agent(
    name="extraction_agent",
    model="gpt-5",
    instructions=instructions
)
```

```python
## manager.py
from .agents.summarization_agent import summarization_agent
from .agents.extraction_agent import extraction_agent

class ServiceManager:
    def run(self, input: ServiceInput) -> ServiceOutput:
        # Select which agent to use
        if input.task_type == "summarization":
            agent = summarization_agent
        elif input.task_type == "extraction":
            agent = extraction_agent
        else:
            raise ValueError(f"Unknown task type: {input.task_type}")

        result = agent.run(input.text)
        return self._format_output(result)
```

Now the manager is clean and focused on orchestration, while agent definitions are modular and reusable.

Exception: Dynamic Modification

In advanced scenarios, you might need to modify an agent's behavior at runtime—for example, adding task-specific context to the agent's instructions. This is permissible, but you should start with a statically defined base agent:

```python
## Advanced case: runtime modification of a static agent
from copy import deepcopy

def run(self, input: ServiceInput) -> ServiceOutput:
    # Start with the static agent definition
    base_agent = summarization_agent

    # Create a modified copy for this specific task
    agent = deepcopy(base_agent)
```

```
    agent.instructions = f"{base_agent.instructions}\n\
nContext: {input.context}"

    result = agent.run(input.text)
    return result
```

This maintains the principle that agent definitions live in `agents/`, while allowing for necessary runtime customization.

Why This Constraint Matters

The prohibition against dynamic agent creation enforces a crucial separation: *orchestration logic* (what happens when) is distinct from *agent behavior* (how individual steps are accomplished). This separation is the foundation of the MUTTA architecture's maintainability and testability benefits.

2.2.3 The Agent Directory: Individual Agent Definitions

The `agents/` directory contains the service's autonomous components—the agents that perform reasoning, make decisions, and use tools to accomplish tasks.

Each agent is defined in its own file, following a consistent structure that makes agents self-documenting and easy to understand in isolation.

System Instructions, Model Selection, and Tool Handoff Assignment

An agent file in the MUTTA architecture typically contains three to four key components:

1. System Instructions

The system instructions (also called the system prompt) define the agent's role, capabilities, and behavior guidelines. These should be clear, specific, and comprehensive.

```
## agents/research_agent.py

instructions = """You are an expert research assistant
    specializing in
academic literature analysis.

Your responsibilities:
- Search for relevant papers using the provided search
    tools
- Evaluate source quality and relevance
- Extract key findings, methodologies, and conclusions
- Maintain academic rigor and cite sources appropriately

Guidelines:
- Prioritize recent publications (last 5 years) unless
    historical context is needed
- Verify claims across multiple sources when possible
- Flag any contradictions or uncertainties in the
    literature
- Organize findings by theme or research question
```

```
17
18 Output format:
19 - Provide structured summaries for each source
20 - Include full citation information
21 - Note confidence level for each finding
22 """
```

Good system instructions are:

- *Specific about the role*: Who is the agent pretending to be?
- *Clear about responsibilities*: What tasks should the agent perform?
- *Explicit about guidelines*: What principles should guide decision-making?
- *Precise about output format*: What structure should the output follow?

2. Model Selection

Different agents in the same service may use different models based on their needs:

- Complex reasoning tasks might need to use recent models with high reasoning capability, such as `gpt-5` or `claude-4.5-sonnet`
- Simple classification tasks might use local models or non-reasoning models like `gpt-5-mini` or `gpt-4.1`
- Cost-sensitive operations might use smaller or local models
- Tasks requiring specific capabilities (e.g., vision) dictate model choice

The model is specified when creating the agent:

```
1 from openai import Agent
2
3 research_agent = Agent(
4     name="research_agent",
5     model="gpt-5",  # Model selection
6     instructions=instructions
7 )
```

Model selection should be documented when non-obvious:

```
1 ## Using GPT 5 for this agent because:
2 # 1. Research tasks require strong reasoning capabilities
3 # 2. Large context window needed for processing multiple
      papers
4 # 3. Quality is more important than cost for this
      application
5
```

```
research_agent = Agent(
    name="research_agent",
    model="gpt-5",
    instructions=instructions
)
```

3. Tool Assignment

If an agent needs access to tools, these are specified in the agent definition:

```
from openai import Agent
from ..tools import search_papers, fetch_pdf,
    extract_citations

research_agent = Agent(
    name="research_agent",
    model="gpt-5",
    instructions=instructions,
    tools=[search_papers, fetch_pdf, extract_citations]
)
```

Tool assignment should be intentional—give agents only the tools they need for their specific responsibilities. An agent with too many tools may become confused about which to use, while an agent with too few cannot accomplish its tasks.

4. Handoff Configuration (Optional)

For multi-agent systems using the handoff pattern (see Section 1.2.2), agents may need the ability to transfer control to specialists:

```
from openai import Agent
from ..agents.citation_specialist import
    citation_specialist
from ..agents.stats_specialist import stats_specialist

research_agent = Agent(
    name="research_agent",
    model="gpt-5",
    instructions=instructions,
    tools=[search_papers, fetch_pdf],
    handoffs=[citation_specialist, stats_specialist]
)
```

A handoff is a special kind of tool that transfers control to another agent. The original agent's context is passed to the specialist, which processes it and returns results.

Complete Agent File Example

Putting it all together, a complete agent file looks like:

```python
## agents/extraction_agent.py
"""
Agent responsible for extracting structured information from documents.
"""

from openai import Agent
from ..tools import highlight_text, lookup_definition

# System instructions define the agent's behavior
instructions = """You are a precise information extraction specialist.

Your task is to extract specific information from documents based on
user-provided schemas. You must:

1. Identify all instances of the requested information types
2. Extract text exactly as it appears (no paraphrasing)
3. Provide context for each extraction (surrounding sentences)
4. Assign confidence scores based on clarity and ambiguity

Tools available:
- highlight_text: Highlights relevant passages in the document
- lookup_definition: Looks up domain-specific terminology

If you encounter ambiguous cases, extract all plausible interpretations
and note the ambiguity in your confidence assessment.

Always output your results using the provided structured output schema.
"""

# Create the agent with model selection and tools
extraction_agent = Agent(
    name="extraction_agent",
    model="gpt-5",  # Requires strong comprehension for accurate extraction
    instructions=instructions,
    tools=[highlight_text, lookup_definition]
)
```

This file is self-contained and declarative. Anyone reading it immediately understands:

- What the agent does (extraction)
- How it should behave (the instructions)
- What model it uses (GPT 5)
- What tools it has access to (highlight and lookup)

Structured Output: Leveraging JSON Schema and Pydantic

One of the most powerful features of modern LLM APIs is structured output—the ability to constrain the model's output to follow a specific JSON schema or Pydantic model. This capability is essential for building reliable services, as it ensures agent outputs are programmatically parseable and type-safe.

Why Structured Output Matters

Without structured output, agent responses are freeform text. Extracting specific information requires fragile string parsing, regular expressions, or additional LLM calls to reformat the output. This approach is:

- *Unreliable*: Text parsing is brittle and fails on unexpected variations
- *Inefficient*: Multiple LLM calls to extract and reformat waste time and money
- *Error-prone*: Parsing failures cause service failures
- *Hard to compose*: Passing unstructured text between agents requires careful prompt engineering

Structured output solves these problems by having the model directly generate valid JSON conforming to a schema.

Defining Output Schemas with Pydantic

Pydantic models provide a Pythonic way to define structured output schemas:

```
from pydantic import BaseModel, Field
from typing import List
from enum import Enum

class ConfidenceLevel(str, Enum):
    HIGH = "high"
    MEDIUM = "medium"
    LOW = "low"

class ExtractedEntity(BaseModel):
    """A single extracted entity from the document."""
    text: str = Field(description="The extracted text, exactly as it appears")
    entity_type: str = Field(description="Type of entity (e.g., 'person', 'date', 'amount')")
    context: str = Field(description="Surrounding text providing context")
    confidence: ConfidenceLevel = Field(description="Extraction confidence level")
    page_number: int = Field(description="Page where the entity was found")

class ExtractionResult(BaseModel):
    """Complete extraction results for a document."""
    entities: List[ExtractedEntity] = Field(description="All extracted entities")
    total_pages: int = Field(description="Total pages processed")
    extraction_complete: bool = Field(description="Whether
```

```
    extraction completed successfully")
23      notes: str = Field(default="", description="Any
    additional notes or warnings")
```

These Pydantic models serve dual purposes:

1. They define the schema that the LLM must follow when generating output
2. They provide type safety and validation in your Python code

Configuring Agents for Structured Output

To use structured output with an agent, specify the response format:

```
from openai import Agent

extraction_agent = Agent(
    name="extraction_agent",
    model="gpt-5",
    instructions=instructions,
    tools=[highlight_text, lookup_definition],
    response_format=ExtractionResult  # Pydantic model
)
```

When you run this agent, the SDK ensures that the model's output conforms to the `ExtractionResult` schema:

```
result = extraction_agent.run(document_text)
# result is automatically parsed into an ExtractionResult
    instance
assert isinstance(result, ExtractionResult)
print(f"Extracted {len(result.entities)} entities")
```

Complete Example: Agent with Structured Output

```
## agents/citation_extractor.py
"""
Agent that extracts citations from academic papers.
"""

from openai import Agent
from pydantic import BaseModel, Field
from typing import List

# Define the output schema
class Citation(BaseModel):
    """A single citation extracted from the paper."""
    authors: List[str] = Field(description="List of author
    names")
    title: str = Field(description="Paper title")
```

```
    year: int = Field(description="Publication year")
    venue: str = Field(description="Journal or conference name")
    citation_text: str = Field(description="Full citation as it appears in the paper")

class CitationExtractionResult(BaseModel):
    """Results of citation extraction."""
    citations: List[Citation] = Field(description="All citations found")
    citation_count: int = Field(description="Total number of citations")

# Define agent instructions
instructions = """You are an expert at extracting citations from academic papers.

Your task:
1. Identify all citations in the reference section
2. Parse each citation to extract: authors, title, year, venue
3. Handle various citation formats (APA, MLA, Chicago, etc.)
4. Preserve the exact citation text as it appears

Be thorough and accurate. If a field is ambiguous or missing, use your best judgment
or leave it empty rather than guessing incorrectly.
"""

# Create agent with structured output
citation_extractor = Agent(
    name="citation_extractor",
    model="gpt-5",
    instructions=instructions,
    response_format=CitationExtractionResult
)
```

When this agent runs, you receive a fully typed, validated `CitationExtractionResult` object that can be directly used in subsequent processing steps.

Benefits of Structured Output in MUTTA Services

In the context of the MUTTA architecture, structured output provides:

- **Clean interfaces between components**: Agents produce typed outputs that utilities and other agents can consume without parsing
- **Reliable service contracts**: The service output schema is enforced, not just documented
- **Easier testing**: You can validate agent outputs against expected schemas
- **Better IDE support**: Type hints enable autocomplete and type checking
- **Automatic validation**: Pydantic validates that outputs conform to the schema, catching errors early

2.2.4 Tools (`tools.py`) and Utilities (`utilities.py`)

The final components of the MUTTA architecture are tools and utilities—helper functions that extend agent capabilities and support orchestration logic, respectively.

Tools: Exclusive Functions for Agent Use

Tools are functions that agents can call to interact with external systems, perform computations, or access information beyond their training data. In the MUTTA architecture, tools are defined in `tools.py` and are specifically designed for agent consumption.

Characteristics of Tools

A well-designed tool:

- **Has a clear, specific purpose**: Each tool does one thing well
- **Includes descriptive documentation**: The docstring explains what the tool does, when to use it, and what it returns
- **Uses type hints**: Parameters and return types are explicitly typed
- **Handles errors gracefully**: Tools should catch expected errors and return informative messages rather than raising exceptions that break agent execution
- **Returns structured data**: When possible, tools return typed objects or dictionaries rather than freeform strings

Defining Tools for OpenAI Agents SDK

In the OpenAI Agents SDK, tools are defined as decorated functions:

```
## tools.py
"""
Tools available to agents in the document analysis service.
"""

from openai import function_tool
from typing import List, Dict
import requests

@function_tool
def search_papers(query: str, max_results: int = 10) ->
    List[Dict[str, str]]:
     """
     Search for academic papers matching the query.

     Args:
         query: Search query string (keywords, authors,
    topics)
         max_results: Maximum number of results to return (
    default: 10)

     Returns:
```

```
        List of papers, each containing title, authors, abstract, and URL

    Example:
        search_papers("machine learning interpretability", max_results=5)
    """
    # Implementation using academic search API
    response = requests.get(
        "https://api.semanticscholar.org/graph/v1/paper/search",
        params={"query": query, "limit": max_results}
    )

    if response.status_code != 200:
        return [{"error": f"Search failed: {response.status_code}"}]

    papers = response.json().get("data", [])
    return [
        {
            "title": paper.get("title", "Unknown"),
            "authors": ", ".join([a.get("name", "") for a in paper.get("authors", [])]),
            "abstract": paper.get("abstract", "No abstract available"),
            "url": paper.get("url", ""),
            "year": paper.get("year", "Unknown")
        }
        for paper in papers
    ]

@function_tool
def extract_text_from_pdf(pdf_url: str) -> str:
    """
    Extract text content from a PDF document.

    Args:
        pdf_url: URL of the PDF to process

    Returns:
        Extracted text content, or error message if extraction fails

    Note:
        This tool handles PDFs up to 50 pages. For larger documents,
        consider using the chunk_pdf tool instead.
    """
    try:
        # Implementation using PDF extraction library
        import pdfplumber
        import io

        response = requests.get(pdf_url)
        pdf_file = io.BytesIO(response.content)
```

```
68
69          text = ""
70          with pdfplumber.open(pdf_file) as pdf:
71              if len(pdf.pages) > 50:
72                  return "Error: PDF too large (>50 pages).
    Use chunk_pdf tool."
73
74              for page in pdf.pages:
75                  text += page.extract_text() + "\n\n"
76
77          return text.strip()
78
79      except Exception as e:
80          return f"Error extracting PDF: {str(e)}"
81
82  @function_tool
83  def calculate_statistics(numbers: List[float]) -> Dict[str,
    float]:
84      """
85      Calculate basic statistics for a list of numbers.
86
87      Args:
88          numbers: List of numerical values
89
90      Returns:
91          Dictionary containing mean, median, std_dev, min,
    max
92
93      Example:
94          calculate_statistics([1.0, 2.0, 3.0, 4.0, 5.0])
95          # Returns: {"mean": 3.0, "median": 3.0, ...}
96      """
97      import statistics
98
99      if not numbers:
100         return {"error": "Empty list provided"}
101
102     return {
103         "mean": statistics.mean(numbers),
104         "median": statistics.median(numbers),
105         "std_dev": statistics.stdev(numbers) if len(numbers
    ) > 1 else 0.0,
106         "min": min(numbers),
107         "max": max(numbers),
108         "count": len(numbers)
109     }
```

Tool Design Best Practices

Descriptive Names: Tool names should clearly indicate their purpose (`search_papers`, not `query`)

Comprehensive Docstrings: The LLM reads the docstring to understand when and how to use the tool. Include purpose, parameters, returns, and examples.

Error Handling: Return error information as data rather than raising exceptions. This allows the agent to see what went wrong and potentially retry or adjust its approach.

Appropriate Scope: Each tool should have a single, clear responsibility. If you find yourself writing a tool that does multiple things, split it into multiple tools.

Deterministic When Possible: While some tools interact with external systems that may change, tools should avoid unnecessary randomness in their behavior.

When to Create a Tool vs. Use an Existing API

Not every function needs to be a custom tool. Many common needs are already addressed by existing tools:

- Web search: Use provided search tools or API integrations
- Code execution: Use code interpreter capabilities
- File operations: Use file system tools

Create custom tools when:

- You need to interact with internal systems or proprietary APIs
- You need to enforce specific business logic or constraints
- Existing tools don't match your requirements
- You want to simplify complex operations into agent-friendly interfaces

Utilities: Deterministic Helpers for Manager Direct Calls

While tools extend agent capabilities, utilities support the manager's orchestration logic. Utilities are deterministic functions that the manager calls directly—agents do not have access to utilities.

Characteristics of Utilities

Utilities differ from tools in key ways:

- **Deterministic**: Given the same input, a utility always produces the same output (no LLM calls, no random behavior)
- **Manager-facing**: Utilities are called by `manager.py`, not by agents
- **Data-focused**: Utilities typically transform, validate, or prepare data
- **Fast**: Utilities should complete quickly, as they're often on the critical path

Common Utility Patterns

Input Validation:

```
## utilities.py
"""
Utility functions for the document analysis service.
"""

from pydantic import ValidationError
from typing import List, Dict
import re

```

```python
def validate_document_input(document_text: str, min_length: int = 100) -> None:
    """
    Validate that document input meets requirements.

    Args:
        document_text: The document to validate
        min_length: Minimum acceptable document length

    Raises:
        ValueError: If validation fails
    """
    if not document_text or not document_text.strip():
        raise ValueError("Document text cannot be empty")

    if len(document_text) < min_length:
        raise ValueError(f"Document too short (min {min_length} characters)")

    # Check for common encoding issues
    try:
        document_text.encode('utf-8')
    except UnicodeEncodeError:
        raise ValueError("Document contains invalid characters")
```

Data Transformation:

```python
def chunk_text(text: str, chunk_size: int = 1000, overlap: int = 100) -> List[str]:
    """
    Split text into overlapping chunks for processing.

    Args:
        text: Text to chunk
        chunk_size: Target size of each chunk in characters
        overlap: Number of characters to overlap between chunks

    Returns:
        List of text chunks
    """
    chunks = []
    start = 0

    while start < len(text):
        end = start + chunk_size
        chunk = text[start:end]
        chunks.append(chunk)
        start = end - overlap

    return chunks

def merge_extractions(extractions: List[Dict]) -> Dict:
```

```
    """
    Merge multiple extraction results, removing duplicates.

    Args:
        extractions: List of extraction dictionaries

    Returns:
        Merged extraction dictionary with deduplicated entries
    """
    merged = {
        "entities": [],
        "seen": set()
    }

    for extraction in extractions:
        for entity in extraction.get("entities", []):
            # Use text as deduplication key
            key = entity["text"].lower().strip()
            if key not in merged["seen"]:
                merged["entities"].append(entity)
                merged["seen"].add(key)

    # Remove internal tracking field
    del merged["seen"]
    return merged
```

Format Conversion:

```
def convert_to_service_output(agent_result, metadata: Dict) -> ServiceOutput:
    """
    Convert agent output to the service's output schema.

    Args:
        agent_result: Raw output from the agent
        metadata: Additional metadata to include

    Returns:
        ServiceOutput instance
    """
    return ServiceOutput(
        data=agent_result.entities,
        total_count=len(agent_result.entities),
        processing_time=metadata.get("duration", 0),
        model_used=metadata.get("model", "unknown"),
        confidence=_calculate_overall_confidence(agent_result)
    )

def _calculate_overall_confidence(result) -> float:
    """Calculate weighted average confidence across all entities."""
    if not result.entities:
```

```
        return 0.0

    confidence_map = {"high": 1.0, "medium": 0.6, "low":
    0.3}
    scores = [confidence_map.get(e.confidence, 0.5) for e
    in result.entities]
    return sum(scores) / len(scores)
```

Why Separate Tools from Utilities?

The distinction between tools and utilities enforces a clean separation of concerns:

- **Tools are for agents**: They extend agent capabilities and are part of the autonomous behavior
- **Utilities are for managers**: They support deterministic orchestration logic

This separation provides clarity: when reading agent definitions, you see what capabilities the agent has (its tools). When reading the manager, you see what deterministic operations support the workflow (utilities).

It also impacts testing: tools must be designed to work with unpredictable agent behavior, while utilities can assume well-formed inputs from the manager.

Complete MUTTA Service Example

Let's conclude this section with a complete, minimal MUTTA service that demonstrates all components working together:

```
## service_example/
# |-- manager.py
# |-- agents/
# |   `-- summarizer.py
# |-- tools.py
# |-- utilities.py
# `-- __init__.py

# --- utilities.py ---
def validate_input(text: str) -> None:
    """Validate input text."""
    if not text or len(text) < 50:
        raise ValueError("Text too short")

def format_output(summary: str, word_count: int) -> dict:
    """Format the service output."""
    return {
        "summary": summary,
        "word_count": word_count,
        "status": "success"
    }

# --- tools.py ---
from openai import function_tool

@function_tool
```

```python
def count_words(text: str) -> int:
    """Count words in text."""
    return len(text.split())

# --- agents/summarizer.py ---
from openai import Agent
from ..tools import count_words

instructions = """You are a summarization expert.
Create concise, accurate summaries of provided text."""

summarizer = Agent(
    name="summarizer",
    model="gpt-5",
    instructions=instructions,
    tools=[count_words]
)

# --- manager.py ---
from .agents.summarizer import summarizer
from .utilities import validate_input, format_output

class SummarizationService:
    """Simple text summarization service."""

    def run(self, text: str) -> dict:
        """
        Summarize input text.

        Args:
            text: Text to summarize

        Returns:
            Dictionary with summary and metadata
        """
        # Use utility for validation
        validate_input(text)

        # Run agent
        summary = summarizer.run(f"Summarize this text:\n\n{text}")

        # Use utility for output formatting
        word_count = len(summary.split())
        return format_output(summary, word_count)

# --- __init__.py ---
from .manager import SummarizationService

__all__ = ["SummarizationService"]
```

This complete example shows:

- Manager orchestrates the flow
- Agent is defined separately with instructions and tools

- Tools are available to the agent
- Utilities support manager operations
- The service has a clean public interface

The MUTTA architecture provides a foundation for building production-grade agent systems. However, architecture alone does not guarantee correctness. In the next chapter, we address a critical challenge: ensuring that the data flowing through our systems is accurate and aligned with each component's expectations. We will explore the "garbage in, garbage out" principle and develop techniques for maximizing input quality at every stage of our agent pipelines.

Chapter 3

Maximizing Input Quality: The "Garbage In, Garbage Out" Principle

In Chapters 1 and 2, we established the foundations of agent systems and introduced the MUTTA architecture for building modular, maintainable services. However, even the most elegant architecture cannot overcome a fundamental limitation: *garbage in, garbage out*.

The quality of a system's output is fundamentally bounded by the quality of its input. This principle, well-known in traditional computing, takes on heightened importance in agent systems. Unlike deterministic algorithms that fail predictably on invalid input, agents attempt to process whatever they receive—often producing plausible-seeming but incorrect outputs when given poor-quality input.

This chapter explores three critical aspects of input quality in agent systems:

1. **Defining "garbage" contextually**: What constitutes good or bad input depends on the specific task and agent capabilities.

2. **Understanding error propagation**: How errors compound across sequential agent steps, and why this places strict bounds on acceptable input quality.

3. **Aligning inputs heuristically**: A practical technique using embeddings to ensure that inputs match agent expectations at every stage of a pipeline.

By the end of this chapter, you will understand not only why input quality matters but also how to measure, validate, and optimize it systematically.

3.1 Understanding Input Accuracy and Relative Garbage

The phrase "garbage in, garbage out" is often invoked as a truism, but what exactly constitutes "garbage" in the context of agent systems? The answer is more nuanced than it might first appear.

3.1.1 Defining 'Garbage' Relative to the Specific Task

In traditional software, invalid input is often clear-cut: a function expecting an integer fails if given a string, a parser rejects malformed JSON, a database query fails on a nonexistent table. The boundaries of validity are sharp and well-defined.

Agent systems are different. An agent will attempt to process almost any input you provide, generating a response even when the input is poorly suited to its task. This adaptability is simultaneously a strength (agents can handle diverse, unexpected inputs) and a vulnerability (they may produce nonsense when inputs are inappropriate).

The Relativity of Input Quality

A crucial insight is that input quality is *relative to the task*. What constitutes good input for one agent may be irrelevant or harmful for another. Consider these examples:

Example 1: Weather Planning Agent

Suppose you have an agent designed to help plan trips based on weather conditions. For this agent:

High-quality input includes:

- User's current or intended location
- Travel dates
- Weather-sensitive preferences (e.g., "I want to avoid rain")

Low-quality or irrelevant input includes:

- Detailed dietary preferences
- Financial information
- Historical political data about the destination

Now consider a different agent—a **recording or note-taking agent** designed to transcribe and organize meeting notes. For this agent:

High-quality input includes:

- Conversation transcript or audio
- Context about the meeting purpose
- Information about participants and their roles

Irrelevant input includes:

- Weather information
- Location coordinates
- Climate forecasts

The weather data that is essential for the first agent is actively harmful for the second—it clutters the context window, distracts from relevant information, and may confuse the agent about its task.

Implications for System Design

This relativity principle has several important implications:

Context-Specific Validation: Input validation must be tailored to each agent's specific task. Generic validation (checking for non-empty strings, valid JSON structure) is necessary but insufficient. You must validate that the input contains

the information your agent needs and excludes irrelevant data that might confuse it.

Tool and Context Selection: When providing tools to an agent, ensure they align with its purpose. A weather agent should have access to weather APIs and location services, not database query tools or code execution environments. Similarly, the context you inject should be task-relevant.

Input Schema Design: When designing input schemas (as discussed in Section 2.1.1), explicitly capture the information your agent needs. Don't accept freeform inputs if you can specify structured requirements.

Progressive Filtering: In multi-agent systems, each stage should filter and refine information, passing forward only what's relevant to downstream agents. This prevents irrelevant data from accumulating as it flows through the pipeline.

Measuring Input Quality

Given that quality is relative, how do we measure it? Several approaches exist:

1. Completeness: Does the input contain all required information?

For a travel planning agent, required information might include destination and dates. You can validate completeness programmatically:

```
def validate_travel_input(input_data: TravelInput) -> None:
    """Validate that travel planning input is complete."""
    required_fields = ["destination", "start_date", "
end_date"]
    missing = [f for f in required_fields if not getattr(
input_data, f, None)]

    if missing:
        raise ValueError(f"Missing required fields: {', '.
join(missing)}")
```

2. Relevance: Does the input focus on task-relevant information?

This is harder to validate programmatically but can be assessed heuristically. One approach is to measure the semantic similarity between the input and the agent's system prompt (a technique we'll formalize in Section 3.3).

3. Clarity: Is the input unambiguous and well-structured?

Clarity can be assessed through:

- Structural validation (for structured inputs)
- Length checks (too short may be underspecified; too long may be unfocused)
- Consistency checks (e.g., dates in logical order)

4. Accuracy: Is the information in the input factually correct?

This is the most challenging dimension to validate, often requiring external verification or domain knowledge. In many cases, verifying input accuracy is as hard as the agent's task itself.

The Diamond Analogy

A useful mental model is to think of inputs as raw materials and outputs as products:

- *Garbage in, garbage out*: Feed your system trash, and it produces trash.
- *Diamonds in, diamonds out*: Feed your system high-quality, relevant, accurate information, and it produces high-quality results.

However, the relationship is not one-to-one. Even with diamond-quality input, a poorly designed agent might produce garbage. Input quality is *necessary* but not *sufficient* for output quality. Conversely, no amount of clever prompting or architecture can fully compensate for fundamentally inadequate input.

Designing for Input Quality

Practical strategies for ensuring input quality:

Explicit Input Contracts: Define and document exactly what your agent needs. Use Pydantic models or JSON schemas to enforce structure:

```
from pydantic import BaseModel, Field, validator
from datetime import datetime

class WeatherTripInput(BaseModel):
    """Input for weather-based trip planning."""

    destination: str = Field(
        description="City or region for trip planning"
    )
    start_date: datetime = Field(
        description="Trip start date"
    )
    end_date: datetime = Field(
        description="Trip end date"
    )
    weather_preferences: str = Field(
        description="Weather-related preferences or constraints"
    )

    @validator('end_date')
    def end_after_start(cls, v, values):
        """Ensure end date is after start date."""
        if 'start_date' in values and v <= values['start_date']:
            raise ValueError('End date must be after start date')
        return v
```

Preprocessing Utilities: Use the utilities component of MUTTA (see Section 2.2.4) to clean and normalize inputs before passing them to agents:

```
def preprocess_document(raw_text: str) -> str:
    """Clean and normalize document text."""
    # Remove excessive whitespace
    text = " ".join(raw_text.split())

```

```
    # Normalize encoding
    text = text.encode('utf-8', errors='ignore').decode('utf-8')

    # Remove non-printable characters
    text = ''.join(char for char in text if char.isprintable() or char.isspace())

    return text.strip()
```

Input Validation Agents: For complex inputs where automated validation is difficult, consider using a specialized validation agent that checks input quality before passing data to the main processing pipeline.

User Guidance: When inputs come from users, provide clear guidance about what information is needed. Use structured forms rather than freeform text fields when possible.

The principle is clear: invest effort in ensuring input quality at the beginning of your pipeline. Every improvement in input quality compounds as it flows through your system, while every deficiency multiplies.

3.2 Error Propagation in Sequential Systems

Having established that input quality is relative and critical, we now explore a more subtle challenge: how errors propagate through sequential agent systems. Understanding error propagation is essential for designing reliable services and for making informed decisions about service depth (recall the Rule of 20 from Section 2.1.2).

3.2.1 Bounding Total System Error by Step Count and Maximum Single-Step Error

Consider a sequential agent system—a pipeline where the output of one agent becomes the input to the next. Even if each individual agent is highly accurate, errors can accumulate as data flows through the pipeline.

A Motivating Example

Suppose you have a three-stage document analysis pipeline:

1. **Extraction Agent**: Extracts key claims from a document (95% accurate)
2. **Verification Agent**: Verifies each claim against sources (95% accurate)
3. **Synthesis Agent**: Synthesizes verified claims into a report (95% accurate)

Each agent has an impressive 95% accuracy rate. However, the *end-to-end* accuracy of the pipeline is not 95%. If we model accuracy as the probability that a step produces no errors, the probability that the entire pipeline produces no errors is:

$$P(\text{no errors}) = 0.95 \times 0.95 \times 0.95 = 0.857$$

The pipeline has only 85.7% accuracy—significantly lower than any individual component. With a 10-step pipeline, accuracy drops to:

$$P(\text{no errors}) = 0.95^{10} \approx 0.599$$

Now we have only 60% end-to-end accuracy despite each step being 95% accurate. This is the fundamental challenge of error propagation in sequential systems.

Formalizing Error Accumulation

For sequential systems, we can bound the total error using a simple but powerful principle. Let:

- n = number of sequential steps
- ϵ_i = error of step i
- $\epsilon_{\max} = \max_i \epsilon_i$ = maximum error across all steps

The total accumulated error E_{total} is bounded by:

$$E_{\text{total}} \leq n \times \epsilon_{\max}$$

This bound applies under certain assumptions (which we'll discuss), but it provides a useful heuristic: *total error grows linearly with the number of steps and is determined by the worst-performing step.*

Intuition from Circuit Theory

This bound is inspired by analogous results in circuit theory. In a series of circuit gates, each introducing some error in its transformation, the total error accumulates. For linear transformations, we can often bound the error precisely. For non-linear transformations, the analysis is more complex but similar principles apply.

In agent systems, each agent invocation can be thought of as a non-linear transformation—the agent takes an input, processes it through the complex non-linear function represented by the LLM, and produces an output. Crucially, if we use the same model for all agents (e.g., GPT 5 throughout), all transformations have the same *structure*, even if they process different data.

This structural similarity allows us to reason about error propagation even in the non-linear case. While individual errors may vary based on input, the *error characteristics* of each step are similar. Thus, we can bound the total error by the number of steps times the maximum error of any single step.

Mathematical Justification

Consider a sequence of transformations $f_1, f_2, \ldots, f_n$, where f_i represents the i-th agent in our pipeline. If x_0 is the initial input, the final output is:

$$y = f_n(f_{n-1}(\cdots f_2(f_1(x_0))\cdots))$$

If each f_i has an associated error ϵ_i, and if we assume that errors are independent (a simplification, but often reasonable), then the total error accumulates additively in the worst case.

For linear systems, we can derive exact bounds. For non-linear systems (like LLMs), the analysis is more complex, but empirically, the linear bound serves as a useful approximation.

Implications for System Design

This error propagation behavior has critical implications:

1. Minimize the Number of Steps

The fewer sequential steps in your pipeline, the less error accumulation. This is one motivation for the Rule of 20 (Section 2.1.2)—beyond 20 steps, error accumulation becomes severe enough that error correction is essential.

Strategies to minimize steps:

- Combine multiple tasks into a single agent call when appropriate
- Use more capable models that can accomplish tasks in fewer steps
- Parallelize independent operations (parallel steps don't accumulate errors sequentially)

2. Improve Your Worst-Performing Step

Since total error is bounded by $n \times \epsilon_{\max}$, improving your worst step has outsized impact. Identify which agent in your pipeline has the highest error rate and focus optimization efforts there:

- Use a more capable model for that step
- Improve the agent's prompts and tools
- Add validation or error correction for that specific step
- Provide better-quality input to that step

3. Bound Individual Step Errors

To bound total error, you must bound individual step errors. Techniques include:

- **Structured outputs**: Force agents to produce validated, structured data (Section 2.2.3), reducing the chance of malformed outputs
- **Output validation**: Add programmatic checks after each step
- **Error correction**: Use verification agents (Chapter 5) to catch and fix errors early

4. Consider Error Correction

When error accumulation becomes unacceptable, add error correction mechanisms. These can:

- Verify outputs against ground truth (when available)
- Use consistency checks to detect errors
- Employ higher-quality models to spot-check lower-quality outputs

- Implement retry logic with refined inputs

We explore error correction patterns in depth in Chapter 5.

The Theoretical Minimum: Hallucination Rate

Even with perfect architecture and perfect input, agent systems face a theoretical minimum error rate: the model's hallucination rate. Language models sometimes generate plausible but incorrect information, and this base error rate cannot be eliminated through architectural improvements alone.

For a perfectly designed system with perfect input:

$$\epsilon_{\text{system}} \geq \epsilon_{\text{hallucination}}$$

This hallucination rate varies by:

- The specific model and its training
- The task domain (models are more accurate in some domains than others)
- The specificity of the query (general questions may elicit more hallucinations than specific ones)

However, in practice, systems are never perfect. Input quality, prompt quality, and architectural decisions all introduce additional error sources. The total error is:

$$\epsilon_{\text{total}} = \epsilon_{\text{hallucination}} + \epsilon_{\text{architecture}} + \epsilon_{\text{input}} + \cdots$$

Our goal is to minimize the controllable components ($\epsilon_{\text{architecture}}$, ϵ_{input}) to approach the theoretical minimum.

3.2.2 The Need for Initial Input Alignment

Given that errors accumulate across sequential steps, and given that total error is bounded by $n \times \epsilon_{\max}$, a critical question emerges: *what is the error of the first step?*

The first step's error depends primarily on:

1. The quality of the initial input
2. The alignment between that input and what the agent expects

If the initial input is poorly aligned with the agent's expectations—even if it's complete, relevant, and accurate—the agent may misinterpret it, leading to errors that propagate through the entire pipeline.

The Alignment Problem

Consider an agent with the following system prompt:

```
1 You are a financial analysis agent. Given a company's
    quarterly
2 financial statement, extract key metrics including revenue,
     expenses,
```

```
profit margin, and year-over-year growth.
```

This agent expects a structured financial statement. Now consider these possible inputs:

Well-aligned input:

```
Q3 2024 Financial Statement
Revenue: $10.5M
Operating Expenses: $6.2M
Net Income: $4.3M
YoY Revenue Growth: 15%
...
```

Poorly aligned input:

```
The company had a great quarter. Sales were up
    significantly,
and we managed to keep costs under control. Overall, we're
    very
pleased with the results.
```

The second input contains relevant information, but its format and structure don't match what the agent expects. The agent may struggle to extract precise metrics, leading to errors.

Even worse, the initial error (struggling with unstructured input) propagates to subsequent agents. If the next agent expects precise metrics and receives uncertain estimates, its own output quality degrades.

Why Initial Alignment Matters Most

The first step in a pipeline is special because:

1. It sets the baseline error for the entire system
2. Its output becomes input to subsequent steps, so its errors propagate fully
3. It's the point where we have the most control (we can validate and preprocess initial inputs more easily than intermediate ones)

Mathematically, if the first step has error ϵ_1, and each subsequent step adds error ϵ_i, the total error is at least:

$$E_{\text{total}} \geq \epsilon_1 + \sum_{i=2}^{n} \epsilon_i$$

Minimizing ϵ_1 directly reduces total error. Moreover, a high-quality first output often leads to lower errors in subsequent steps, as agents can work with better inputs.

Validation vs. Alignment

It's important to distinguish *validation* from *alignment*:
Validation checks that input meets basic requirements:

- Is the input non-empty?
- Does it conform to the expected schema?
- Does it contain required fields?

Alignment checks that input matches the agent's expectations:

- Is the input formatted as the agent expects?
- Does the input emphasize information the agent considers important?
- Is the input's structure and style consistent with what the agent is designed to process?

Validation is necessary but insufficient. An input can pass validation yet be poorly aligned with the agent's capabilities and expectations.

In the next section, we develop a practical heuristic for measuring and improving input alignment using embeddings.

3.3 The Input Alignment Heuristic Using Embeddings

We've established that input alignment is critical and that errors accumulate across sequential steps. Now we present a practical technique for ensuring that inputs are well-aligned with agent expectations: the **embedding-based input alignment heuristic**.

This heuristic leverages embedding models—the same technology that underlies retrieval-augmented generation (RAG) and semantic search—to measure how well an input matches what an agent is designed to process.

3.3.1 Generating the Expected Input for the Agent

The first step in the alignment heuristic is to characterize what constitutes a "perfect" input for your agent. Rather than manually specifying this (which is error-prone and incomplete), we use the agent itself to generate examples.

The Process

1. Take your agent's system prompt
2. Ask a language model: "Given this agent's role and instructions, what would be an ideal input?"
3. Have the model generate one or more example inputs
4. These examples represent the agent's "expected input distribution"

Implementation

```python
from openai import OpenAI

def generate_expected_input(system_prompt: str, num_examples: int = 3) -> list[str]:
    """
    Generate example inputs that would be ideal for an agent with
    the given system prompt.

    Args:
        system_prompt: The agent's system instructions
        num_examples: Number of example inputs to generate

    Returns:
        List of example input strings
    """
    client = OpenAI()

    prompt = f"""Given an agent with the following system prompt:

{system_prompt}

Generate {num_examples} examples of ideal inputs for this agent.
Each input should be well-structured, complete, and perfectly suited
to the agent's capabilities and expectations.

Return only the example inputs, one per line, without additional commentary.
"""

    response = client.chat.completions.create(
        model="gpt-5",
        messages=[{"role": "user", "content": prompt}]
    )

    examples = response.choices[0].message.content.strip().split('\n')
    return [ex.strip() for ex in examples if ex.strip()]
```

Example: Financial Analysis Agent

For the financial analysis agent from earlier, generated expected inputs might look like:

```
Example 1:
Q3 2024 Financial Statement
Revenue: $10.5M
Operating Expenses: $6.2M
Net Income: $4.3M
YoY Revenue Growth: 15%
```

```
Profit Margin: 41%

Example 2:
Quarterly Financial Report - Q2 2024
Total Revenue: $8.7M (YoY: +12%)
Cost of Goods Sold: $3.2M
Operating Expenses: $4.1M
Net Profit: $1.4M
Profit Margin: 16%

Example 3:
[Company Name] Financial Summary Q1 2024
Sales Revenue: $12.3M
Expenses: $8.9M
Net Income: $3.4M
Year-over-Year Growth: 18%
Current Profit Margin: 28%
```

These examples capture the expected structure, terminology, and level of detail. They represent what the agent is "tuned" to process effectively.

Why This Works

This approach works because:

- The system prompt encodes the agent's expectations
- Language models can extrapolate from descriptions to examples
- Multiple generated examples capture variation in acceptable formats
- The process is automated and consistent

Refinement

In practice, you may want to:

- Review and curate generated examples
- Add domain-specific examples manually
- Generate examples for different input scenarios (edge cases, minimal inputs, complex inputs)
- Regenerate examples when the agent's system prompt changes

3.3.2 Creating the Guide Embedding (Averaging Expected Input and System Prompt)

With expected inputs generated, we now create a *guide embedding*—a vector representation that captures what well-aligned input "looks like" for this agent.

Embeddings Primer

Embedding models convert text into high-dimensional vectors (typically 1024-1536 dimensions) that capture semantic meaning. Texts with similar meanings have vectors that are close together in this embedding space, measured by cosine similarity.

For example:

- "The cat sat on the mat" and "A feline rested on the rug" have high similarity
- "The cat sat on the mat" and "Quantum physics equations" have low similarity

Creating the Guide Embedding
To create a guide embedding for our agent:

1. Embed the agent's system prompt
2. Embed each expected input example
3. Compute the average (centroid) of all these embeddings

The resulting average embedding represents the "center" of the well-aligned input space.

Implementation

```python
import numpy as np
from openai import OpenAI

def create_guide_embedding(
    system_prompt: str,
    expected_inputs: list[str],
    model: str = "text-embedding-3-small"
) -> np.ndarray:
    """
    Create a guide embedding representing well-aligned inputs.

    Args:
        system_prompt: The agent's system instructions
        expected_inputs: List of example ideal inputs
        model: Embedding model to use

    Returns:
        Guide embedding as a numpy array
    """
    client = OpenAI()

    # Collect all texts to embed
    texts_to_embed = [system_prompt] + expected_inputs

    # Get embeddings
    response = client.embeddings.create(
        model=model,
        input=texts_to_embed
    )

    # Extract embedding vectors
    embeddings = [np.array(item.embedding) for item in response.data]

    # Compute average
    guide_embedding = np.mean(embeddings, axis=0)

```

```
    # Normalize (optional but recommended for cosine similarity)
    guide_embedding = guide_embedding / np.linalg.norm(guide_embedding)

    return guide_embedding
```

Why Average System Prompt and Expected Inputs?
The guide embedding combines two sources of information:

- **System prompt**: Represents the agent's role, responsibilities, and behavioral expectations
- **Expected inputs**: Represent the structure and content of ideal inputs

By averaging, we create a single embedding that captures both "what the agent does" and "what good input looks like." This combined representation serves as our alignment target.

Weighting (Advanced)
In some cases, you may want to weight the system prompt differently from the expected inputs:

```
## Weight system prompt more heavily (e.g., 2x weight)
guide_embedding = (
    2 * system_prompt_embedding +
    sum(input_embeddings)
) / (2 + len(input_embeddings))
```

Experiment with weights to find what works best for your specific use case.

3.3.3 Matching the User Prompt Template against the Guide Embedding

With a guide embedding in hand, we can now measure how well any given input aligns with the agent's expectations.

The Matching Process

1. Take your input template (the format in which users provide inputs)
2. Embed the template
3. Compute cosine similarity between the template embedding and the guide embedding
4. Use this similarity score as an alignment metric

Cosine Similarity
Cosine similarity measures the angle between two vectors, ranging from -1 (opposite) to 1 (identical):

$$\text{cosine\_similarity}(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|}$$

For normalized vectors (as we created above), this simplifies to the dot product.

Implementation

```python
def measure_input_alignment(
    input_template: str,
    guide_embedding: np.ndarray,
    model: str = "text-embedding-3-small"
) -> float:
    """
    Measure how well an input template aligns with agent expectations.

    Args:
        input_template: The input format to evaluate
        guide_embedding: The guide embedding for this agent
        model: Embedding model to use

    Returns:
        Alignment score (0 to 1, higher is better)
    """
    client = OpenAI()

    # Embed the input template
    response = client.embeddings.create(
        model=model,
        input=[input_template]
    )

    template_embedding = np.array(response.data[0].embedding)
    template_embedding = template_embedding / np.linalg.norm(template_embedding)

    # Compute cosine similarity
    similarity = np.dot(guide_embedding, template_embedding)

    return float(similarity)
```

Interpreting Alignment Scores

Alignment scores typically range from 0.5 to 0.95 for real inputs:

- **0.85-0.95**: Excellent alignment, input matches expectations well
- **0.75-0.85**: Good alignment, minor mismatches
- **0.65-0.75**: Moderate alignment, input may need restructuring
- $<$ **0.65**: Poor alignment, input likely unsuitable for this agent

These thresholds are heuristic and should be calibrated for your specific use case.

Using Alignment Scores

You can use alignment scores in several ways:

1. Validation Gate

Reject inputs that fall below a threshold:

```
def validate_input_alignment(
    user_input: str,
    guide_embedding: np.ndarray,
    threshold: float = 0.70
) -> None:
    """
    Validate that input is sufficiently aligned, or raise
    an error.

    Args:
        user_input: The user's input
        guide_embedding: Guide embedding for the agent
        threshold: Minimum acceptable alignment score

    Raises:
        ValueError: If alignment is below threshold
    """
    score = measure_input_alignment(user_input,
    guide_embedding)

    if score < threshold:
        raise ValueError(
            f"Input alignment too low ({score:.3f} < {
    threshold}). "
            f"Please provide input in the expected format."
        )
```

2. User Feedback

Provide feedback to users about input quality:

```
score = measure_input_alignment(user_input, guide_embedding
    )

if score < 0.70:
    print("Warning: Your input may not match what this
    agent expects.")
    print("Consider restructuring to include: [key fields]"
    )
elif score < 0.80:
    print("Your input is acceptable but could be improved."
    )
else:
    print("Your input looks great!")
```

3. Template Optimization

When designing input templates, test multiple versions and select the one with highest alignment:

```
templates = [
    "Financial statement with revenue: {revenue}, expenses: {expenses}",
    "Q{quarter} {year}: Revenue ${revenue}, Expenses ${expenses}",
    "Company financials - Rev: {revenue}, Exp: {expenses}"
]

scores = [
    measure_input_alignment(template, guide_embedding)
    for template in templates
]

best_template = templates[np.argmax(scores)]
print(f"Best template: {best_template} (score: {max(scores):.3f})")
```

Complete Example: Financial Analysis Agent

```
## 1. Define agent system prompt
system_prompt = """You are a financial analysis agent. Given a company's
quarterly financial statement, extract key metrics including revenue,
expenses, profit margin, and year-over-year growth."""

# 2. Generate expected inputs
expected_inputs = generate_expected_input(system_prompt, num_examples=3)

# 3. Create guide embedding
guide = create_guide_embedding(system_prompt, expected_inputs)

# 4. Test different input formats
test_inputs = [
    "Q3 2024: Revenue $10.5M, Expenses $6.2M, YoY Growth 15%",  # Good format
    "The company made $10.5M in Q3 and spent $6.2M",  # Informal
    "Great quarter! Sales up 15%, looking strong"  # Very informal
]

print("Alignment scores:")
for inp in test_inputs:
    score = measure_input_alignment(inp, guide)
    print(f"{score:.3f}: {inp[:50]}...")
```

```
23
24 # Output:
25 # 0.876: Q3 2024: Revenue $10.5M, Expenses $6.2M, YoY Gro
       ...
26 # 0.782: The company made $10.5M in Q3 and spent $6.2M...
27 # 0.623: Great quarter! Sales up 15%, looking strong...
```

The alignment scores clearly differentiate between well-formatted and poorly-formatted inputs, even though all three contain similar information.

3.3.4 Recursive Application: Maximizing Internal Agent Communications

The alignment heuristic becomes even more powerful when applied *recursively* throughout a multi-agent pipeline. Instead of only validating the initial input, we validate that each agent's output is well-aligned with the next agent's expectations.

The Problem of Inter-Agent Misalignment

In a multi-agent pipeline, each agent has its own system prompt and expectations. Even if Agent A produces high-quality output, that output may not match what Agent B expects as input.

Consider this pipeline:

1. **Extraction Agent**: Extracts financial data from unstructured text
2. **Analysis Agent**: Analyzes financial data to compute metrics
3. **Report Agent**: Generates a formatted report from the analysis

The Extraction Agent might output JSON, the Analysis Agent might expect CSV, and the Report Agent might expect Markdown. Without explicit attention to inter-agent alignment, mismatches are likely.

Recursive Alignment Process

For each transition between agents in your pipeline:

1. Create a guide embedding for the receiving agent (as described above)
2. Measure the alignment between the sending agent's output format and the receiving agent's guide
3. If alignment is low, add a transformation step or restructure one of the agents

Implementation in Manager

```
1 class PipelineManager:
2     """Manager with recursive alignment checking."""
3
4     def __init__(self):
5         # Precompute guide embeddings for all agents
6         self.guide_embeddings = {
7             "extractor": create_guide_embedding(
```

```
                extractor_prompt,
                generate_expected_input(extractor_prompt)
            ),
            "analyzer": create_guide_embedding(
                analyzer_prompt,
                generate_expected_input(analyzer_prompt)
            ),
            "reporter": create_guide_embedding(
                reporter_prompt,
                generate_expected_input(reporter_prompt)
            )
        }

    def run(self, user_input: str) -> str:
        """Run pipeline with alignment validation at each stage."""

        # Validate initial input
        self._validate_alignment(
            user_input,
            self.guide_embeddings["extractor"],
            stage="initial"
        )

        # Run extraction
        extracted = extractor_agent.run(user_input)

        # Validate extraction output against analyzer expectations
        self._validate_alignment(
            str(extracted),
            self.guide_embeddings["analyzer"],
            stage="extraction->analysis"
        )

        # Run analysis
        analyzed = analyzer_agent.run(extracted)

        # Validate analysis output against reporter expectations
        self._validate_alignment(
            str(analyzed),
            self.guide_embeddings["reporter"],
            stage="analysis->report"
        )

        # Run reporting
        report = reporter_agent.run(analyzed)

        return report

    def _validate_alignment(
        self,
        data: str,
        guide: np.ndarray,
        stage: str,
```

```
        threshold: float = 0.70
    ) -> None:
        """Validate alignment with warning/error."""
        score = measure_input_alignment(data, guide)

        if score < threshold:
            print(f"WARNING: Low alignment at {stage}: {score:.3f}")
            # Could raise error in production
```

Designing for Alignment

When building multi-agent pipelines, explicitly design for inter-agent alignment:

1. Standardize Output Formats

Use consistent structured output formats (Pydantic models) across agents:

```
## All agents in the pipeline use similar structured outputs
class Metadata(BaseModel):
    source: str
    timestamp: str
    confidence: float

class ExtractionData(BaseModel):
    entities: list[str]
    claims: list[str]

class ExtractionOutput(BaseModel):
    data: ExtractionData
    metadata: Metadata

class AnalysisData(BaseModel):
    findings: list[str]
    conclusions: list[str]

class AnalysisOutput(BaseModel):
    data: AnalysisData  # Nested Pydantic model
    metadata: Metadata
```

2. Explicit Transformation Steps

When agents have incompatible formats, add explicit transformation utilities:

```
def transform_extraction_to_analysis_format(
    extraction: ExtractionOutput
) -> str:
    """Transform extraction output to match analyzer's expected input."""
    # Convert from agent A's format to agent B's format
    return format_for_analysis(extraction.data)
```

3. Iterative Refinement

During development, use alignment scores to identify mismatches:

1. Build initial pipeline
2. Measure alignment at each transition
3. Identify lowest-scoring transitions
4. Refine agent prompts or add transformations
5. Re-measure until all transitions have acceptable alignment

Benefits of Recursive Alignment
Applying alignment checking throughout your pipeline:

- **Catches mismatches early**: Detects format incompatibilities during development
- **Reduces error propagation**: Well-aligned inter-agent communication minimizes the ϵ_i for each step
- **Guides refactoring**: Alignment scores tell you which agent boundaries need work
- **Documents expectations**: Guide embeddings serve as implicit documentation of what each agent expects

Monitoring in Production
In production systems, log alignment scores for monitoring:

```python
import logging

def run_with_monitoring(self, user_input: str) -> str:
    """Run pipeline with alignment monitoring."""

    # Log initial alignment
    initial_score = measure_input_alignment(
        user_input,
        self.guide_embeddings["extractor"]
    )
    logging.info(f"Initial alignment: {initial_score:.3f}")

    # ... run pipeline ...

    # Log inter-agent alignments
    # Alert if scores drop below thresholds
```

This monitoring helps detect:

- User inputs that don't match expectations (suggesting need for better guidance)
- Agent output drift (if an agent's behavior changes over time)

- System degradation (if alignment scores gradually decrease)

The alignment heuristic gives us a deterministic, automatable way to minimize ϵ_1—the error of the first step—and to validate that each agent in a pipeline receives inputs that match its expectations. Combined with the MUTTA architecture from Chapter 2, this provides a solid foundation for building reliable agent services.

However, even with perfect alignment and minimal input error, agent systems face challenges from hallucinations, tool failures, and unexpected edge cases. In Chapter 4, we'll explore reusable agent patterns like RAG and the Navigator that provide agents with better access to information, reducing errors at the source. Then in Chapter 5, we'll develop comprehensive error correction and verification strategies to catch and correct errors that slip through despite our best efforts.

The journey from raw inputs to reliable outputs requires attention at every stage. Input quality and alignment form the foundation; the patterns and error correction strategies we'll explore next build upon this foundation to create production-ready agent systems.

Chapter 4

Universal Reusable Agent Patterns

In the previous chapters, we established the foundations of agent systems (Chapter 1), introduced the MUTTA architecture for building modular services (Chapter 2), and developed techniques for ensuring input quality and alignment (Chapter 3). These principles provide the structural foundation for building robust agent systems.

However, structure alone does not solve the fundamental challenges that agents face in accomplishing complex tasks. Agents are limited by their training data, context windows, and inherent capabilities. To build truly powerful systems, we must equip agents with patterns that extend their reach beyond these limitations.

This chapter introduces four **universal reusable agent patterns**—architectural patterns that can be applied across virtually any agent system to enhance capabilities, reduce errors, and enable new classes of functionality:

1. **Retrieval Augmented Generation (RAG)**: Overcoming knowledge limitations by retrieving relevant information from external sources
2. **The Navigator Pattern**: Exploring structured and unstructured data spaces to find contextually relevant information
3. **Code Interpreter**: Providing unlimited problem-solving capability through arbitrary code execution
4. **Tool Selector**: Managing large tool inventories that exceed context window capacity

These patterns are called "universal" because they address fundamental limitations that apply across domains and use cases. They are "reusable" because, once implemented, they can be deployed in multiple agents and services with minimal modification. Most importantly, incorporating these patterns into your architecture almost always improves system quality—they rarely degrade performance and often provide transformative capability enhancements.

Understanding these patterns is essential for any practitioner building production agent systems. They represent battle-tested solutions to common challenges and form the toolkit upon which sophisticated agentic applications are built.

4.1 Retrieval Augmented Generation (RAG)

The first and perhaps most fundamental reusable pattern is Retrieval Augmented Generation, commonly known as RAG. This pattern addresses a critical limitation of language models: they are trained on fixed datasets and cannot access information beyond their training cutoff or domain-specific knowledge not present in their training corpus.

4.1.1 RAG for Solving Knowledge Gaps and Reducing Hallucination

Language models are impressive in their breadth of knowledge, but they have clear boundaries. A model trained on data through 2023 knows nothing about events in 2024. A general-purpose model may have limited knowledge about proprietary systems, niche technical domains, or organization-specific information. When asked about topics outside their knowledge, models have two problematic behaviors:

1. **Declining to answer**: The model states it doesn't have the information (better than hallucinating, but not helpful)

2. **Hallucinating**: The model generates plausible-sounding but incorrect information (dangerous and difficult to detect)

RAG solves this problem by augmenting the model's context with retrieved information. Instead of relying solely on parametric knowledge (encoded in the model's weights), the agent can access non-parametric knowledge (stored in external databases, documents, or knowledge bases).

The Core Idea

At its heart, RAG is simple:

1. Maintain a knowledge base of relevant documents or information chunks

2. When a query arrives, retrieve the most relevant pieces of information

3. Inject the retrieved information into the agent's context

4. The agent generates a response grounded in the retrieved information

This pattern is analogous to a human consulting reference materials before answering a question. Just as you might look up specifications in a manual before answering a technical question, the agent retrieves relevant documents before generating its response.

Why RAG Reduces Hallucination

Hallucinations occur when models generate information without grounding in actual knowledge. By providing relevant, factual information in the context, RAG:

- **Anchors responses to facts**: The model can cite specific retrieved information rather than generating from its potentially faulty parametric knowledge

- **Provides recent information**: Retrieved documents can contain up-to-date information beyond the model's training cutoff

- **Enables verification**: Users can check the retrieved sources to validate the agent's claims

- **Reduces uncertainty**: When the model has concrete information to work with, it's less likely to make things up

Use Cases for RAG

RAG is valuable whenever agents need to work with:

- **Domain-specific knowledge**: Technical documentation, medical literature, legal precedents, scientific papers
- **Organization-specific information**: Internal wikis, policy documents, product specifications, customer data
- **Recent information**: News articles, updated regulations, current events
- **Large knowledge bases**: Any information corpus too large to fit in a single context window
- **Frequently updated content**: Information that changes regularly and would make fine-tuning impractical

Example: Game Strategy Assistant

Consider building an agent that helps players optimize strategies in a video game. The agent needs to know:

- Character abilities and stats
- Item effects and combinations
- Map layouts and strategic positions
- Optimal combo sequences
- Patch notes and balance changes

A general-purpose language model likely has limited or no knowledge about a proprietary or niche game. Fine-tuning the model on game data is possible but expensive and inflexible (every game update would require retraining). RAG provides an elegant solution: maintain a knowledge base of game information and retrieve relevant details for each query.

When a player asks "What's the best build for a tank character?", the agent:

1. Retrieves documents about tank characters, defensive items, and survivability strategies
2. Uses this concrete information to provide specific, accurate recommendations
3. Can cite specific stats and mechanics from the retrieved documents

4.1.2 Mechanism: Embedding Documentation and Cosine Search for Context Injection

Now that we understand why RAG is valuable, let's explore how it works mechanically. The implementation relies on embedding models and semantic search—the same technologies we used for input alignment in Chapter 3.

Step 1: Chunking and Embedding Documents

The first step is preparing your knowledge base:

1a. Chunk Documents into Atomic Units

Large documents must be broken into smaller chunks. The chunk size represents a trade-off:

- *Smaller chunks* (e.g., paragraphs): More precise retrieval, but may lack context
- *Larger chunks* (e.g., sections): More context, but may include irrelevant information

Typical chunk sizes range from 200-1000 tokens, depending on your domain and needs.

```python
def chunk_document(document: str, chunk_size: int = 500, overlap: int = 50) -> list[str]:
    """
    Split a document into overlapping chunks.

    Args:
        document: Full document text
        chunk_size: Target chunk size in characters
        overlap: Overlap between chunks to preserve context

    Returns:
        List of document chunks
    """
    words = document.split()
    chunks = []

    start = 0
    while start < len(words):
        # Take chunk_size words
        end = min(start + chunk_size, len(words))
        chunk = " ".join(words[start:end])
        chunks.append(chunk)

        # Move forward by (chunk_size - overlap)
        start += (chunk_size - overlap)

    return chunks
```

What Makes a Good Chunk?
Good chunks are:

- **Semantically coherent**: Each chunk should represent a complete idea or concept
- **Self-contained**: The chunk should be understandable without requiring the full document
- **Appropriately sized**: Large enough to be useful, small enough to be precise
- **Overlapping**: Slight overlap ensures important information at chunk boundaries isn't lost

For structured documents (like technical documentation with clear sections), respect natural boundaries:

```python
def chunk_by_sections(document: str, max_chunk_size: int = 1000) -> list[dict]:
    """
    Chunk a document by its section structure.

    Args:
        document: Markdown or structured document
        max_chunk_size: Maximum chunk size; split large sections if needed

    Returns:
        List of chunks with metadata
    """
    import re

    # Split by headers (e.g., ## or ###)
    sections = re.split(r'\n##+ ', document)

    chunks = []
    for section in sections:
        if not section.strip():
            continue

        # Extract section title
        lines = section.split('\n', 1)
        title = lines[0].strip()
        content = lines[1] if len(lines) > 1 else ""

        # If section is too large, sub-chunk it
        if len(content) > max_chunk_size:
            sub_chunks = chunk_document(content, chunk_size=max_chunk_size // 4)
            for i, sub_chunk in enumerate(sub_chunks):
                chunks.append({
                    "text": sub_chunk,
                    "title": f"{title} (part {i+1})",
                    "section": title
                })
        else:
            chunks.append({
                "text": content,
                "title": title,
                "section": title
            })

    return chunks
```

1b. Embed Each Chunk

Once you have chunks, embed each one using an embedding model:

```python
from openai import OpenAI
import numpy as np
```

```
3
4  def embed_chunks(chunks: list[str], model: str = "text-
       embedding-3-small") -> list[np.ndarray]:
5      """
6      Generate embeddings for document chunks.
7
8      Args:
9          chunks: List of text chunks
10         model: Embedding model to use
11
12     Returns:
13         List of embedding vectors
14     """
15     client = OpenAI()
16
17     # Batch embed for efficiency (OpenAI supports up to
       2048 inputs per request)
18     batch_size = 2048
19     all_embeddings = []
20
21     for i in range(0, len(chunks), batch_size):
22         batch = chunks[i:i + batch_size]
23
24         response = client.embeddings.create(
25             model=model,
26             input=batch
27         )
28
29         embeddings = [np.array(item.embedding) for item in
       response.data]
30         all_embeddings.extend(embeddings)
31
32     return all_embeddings
```

1c. Store Chunks and Embeddings

Store the chunks and their embeddings for later retrieval. For simple applications, an in-memory structure suffices:

```
1  class SimpleRAGStore:
2      """Simple in-memory RAG knowledge base."""
3
4      def __init__(self):
5          self.chunks = []
6          self.embeddings = []
7          self.metadata = []
8
9      def add_document(self, document: str, metadata: dict =
       None):
10         """Add a document to the knowledge base."""
11         # Chunk the document
12         chunks = chunk_document(document)
13
14         # Embed chunks
```

```
        embeddings = embed_chunks(chunks)

        # Store
        for chunk, embedding in zip(chunks, embeddings):
            self.chunks.append(chunk)
            self.embeddings.append(embedding)
            self.metadata.append(metadata or {})

    def save(self, filepath: str):
        """Save the knowledge base to disk."""
        import pickle
        with open(filepath, 'wb') as f:
            pickle.dump({
                'chunks': self.chunks,
                'embeddings': self.embeddings,
                'metadata': self.metadata
            }, f)

    def load(self, filepath: str):
        """Load the knowledge base from disk."""
        import pickle
        with open(filepath, 'rb') as f:
            data = pickle.load(f)
            self.chunks = data['chunks']
            self.embeddings = data['embeddings']
            self.metadata = data['metadata']
```

For production systems with large knowledge bases, use a vector database like Pinecone, Weaviate, or Qdrant.

Step 2: Query-Time Retrieval

When a user query arrives, retrieve the most relevant chunks:

2a. Embed the Query

```
def embed_query(query: str, model: str = "text-embedding-3-small") -> np.ndarray:
    """Embed a user query."""
    client = OpenAI()

    response = client.embeddings.create(
        model=model,
        input=[query]
    )

    return np.array(response.data[0].embedding)
```

2b. Compute Similarity and Retrieve Top-K

Use cosine similarity (as in Chapter 3) to find the most relevant chunks:

```
class SimpleRAGStore:
    # ... previous methods ...
```

```
3
4       def retrieve(self, query: str, top_k: int = 5) -> list[
        dict]:
5           """
6           Retrieve the top-k most relevant chunks for a query
        .
7
8           Args:
9               query: User query
10              top_k: Number of chunks to retrieve
11
12          Returns:
13              List of dicts with 'text', 'score', and '
        metadata'
14          """
15          # Embed the query
16          query_embedding = embed_query(query)
17
18          # Normalize query embedding
19          query_embedding = query_embedding / np.linalg.norm(
        query_embedding)
20
21          # Compute cosine similarity with all chunks
22          similarities = []
23          for i, chunk_embedding in enumerate(self.embeddings
        ):
24              # Normalize chunk embedding
25              norm_chunk = chunk_embedding / np.linalg.norm(
        chunk_embedding)
26
27              # Cosine similarity (dot product of normalized
        vectors)
28              similarity = np.dot(query_embedding, norm_chunk
        )
29
30              similarities.append({
31                  'text': self.chunks[i],
32                  'score': float(similarity),
33                  'metadata': self.metadata[i],
34                  'index': i
35              })
36
37          # Sort by similarity (descending) and take top-k
38          similarities.sort(key=lambda x: x['score'], reverse
        =True)
39          return similarities[:top_k]
```

Step 3: Context Injection

Inject the retrieved chunks into the agent's context:

```
1   def create_rag_context(query: str, retrieved_chunks: list[
        dict]) -> str:
2       """
```

```
    Create an augmented context from retrieved chunks.

    Args:
        query: User's query
        retrieved_chunks: Retrieved relevant chunks

    Returns:
        Formatted context string
    """
    context_parts = ["# Retrieved Context\n"]

    for i, chunk in enumerate(retrieved_chunks, 1):
        context_parts.append(f"## Source {i} (relevance: {chunk['score']:.3f})")
        context_parts.append(chunk['text'])
        context_parts.append("")  # Blank line

    context_parts.append("# User Query")
    context_parts.append(query)

    return "\n".join(context_parts)
```

Step 4: Agent Generation with RAG

Finally, the agent generates a response using the augmented context:

```
from openai import Agent

# Define RAG-enabled agent
rag_agent_instructions = """You are a helpful assistant with access to
retrieved context. When answering questions:

1. Ground your responses in the provided retrieved context
2. Cite specific sources when making claims
3. If the context doesn't contain relevant information, say so
4. Don't make up information beyond what's in the context
"""

rag_agent = Agent(
    name="rag_agent",
    model="gpt-5",
    instructions=rag_agent_instructions
)

def answer_with_rag(query: str, knowledge_base: SimpleRAGStore) -> str:
    """
    Answer a query using RAG.

    Args:
        query: User's question
        knowledge_base: RAG knowledge base
```

```
    Returns:
        Agent's response grounded in retrieved context
    """
    # Retrieve relevant chunks
    retrieved = knowledge_base.retrieve(query, top_k=5)

    # Create augmented context
    augmented_context = create_rag_context(query, retrieved
    )

    # Generate response
    response = rag_agent.run(augmented_context)

    return response
```

Complete RAG Example: Game Strategy Assistant

Putting it all together:

```
## Initialize knowledge base
game_kb = SimpleRAGStore()

# Add game documentation
game_kb.add_document("""
# Tank Characters

Tank characters specialize in survivability and crowd
    control.

## Best Items for Tanks
- Reinforced Armor: +500 HP, +50 Defense
- Guardian's Shield: Blocks 30% of incoming damage
- Health Regeneration Amulet: Restores 2% HP per second

## Optimal Build
1. Start with Reinforced Armor for early game survivability
2. Add Guardian's Shield mid-game
3. Complete with Health Regeneration Amulet
4. Consider situational items based on enemy composition
""", metadata={"source": "tank_guide.md"})

game_kb.add_document("""
# Damage Dealers

Damage dealers focus on eliminating enemies quickly.

## Best Items for Damage
- Critical Strike Dagger: +75 Attack, 20% crit chance
- Berserker's Axe: +100 Attack, stacking damage buff
- Shadow Cloak: +50 Attack, invisibility ability
""", metadata={"source": "damage_guide.md"})

# Use RAG to answer questions
```

```
query = "What's the best build for a tank character?"
answer = answer_with_rag(query, game_kb)

print(answer)
# Output will reference the specific items and build order from the
# retrieved tank documentation, rather than hallucinating items.
```

RAG Best Practices

1. Chunk Size Optimization

Experiment with different chunk sizes for your domain. Run evaluations:

```
chunk_sizes = [200, 500, 1000]
for size in chunk_sizes:
    # Build RAG system with this chunk size
    # Test on benchmark queries
    # Measure accuracy and relevance
    # Select the size with best performance
```

2. Retrieval Quality Matters

The quality of your RAG system depends critically on retrieval quality. Poor retrieval leads to irrelevant context, confusing the agent. Monitor retrieval quality:

```
def evaluate_retrieval(kb: SimpleRAGStore, test_cases: list[dict]):
    """
    Evaluate retrieval quality on test cases.

    Args:
        kb: Knowledge base
        test_cases: List of {'query': str, 'expected_doc_ids': list[int]}
    """
    for case in test_cases:
        retrieved = kb.retrieve(case['query'], top_k=5)
        retrieved_ids = [r['index'] for r in retrieved]

        # Check if expected documents are in top-k
        hits = sum(1 for exp_id in case['expected_doc_ids']
                   if exp_id in retrieved_ids)
        recall = hits / len(case['expected_doc_ids'])

        print(f"Query: {case['query']}")
        print(f"Recall@5: {recall:.2f}")
```

3. Hybrid Search

For even better retrieval, combine semantic search (embeddings) with keyword search:

- Semantic search finds conceptually similar content
- Keyword search finds exact term matches
- Combining both improves robustness

4. Metadata Filtering
Store metadata with chunks and filter before retrieval:

```python
def retrieve_filtered(
    self,
    query: str,
    top_k: int = 5,
    metadata_filter: dict = None
) -> list[dict]:
    """Retrieve with metadata filtering."""
    # First filter by metadata
    if metadata_filter:
        valid_indices = [
            i for i, meta in enumerate(self.metadata)
            if all(meta.get(k) == v for k, v in
    metadata_filter.items())
        ]
    else:
        valid_indices = list(range(len(self.chunks)))

    # Then retrieve from filtered set
    # ... (similarity computation on valid_indices only)
```

This allows queries like "What tank items are good?" with metadata filter ‘"category": "tanks"‘.

RAG in the MUTTA Architecture
In MUTTA, RAG is typically implemented as:

- **Knowledge base**: Initialized in manager’s `__init__` or loaded from disk
- **Retrieval utility**: A utility function in `utilities.py` that retrieves relevant chunks
- **RAG-enabled agents**: Agents in `agents/` directory that expect augmented context

The manager orchestrates: retrieve → augment context → call agent.

4.2 The Navigator Pattern (Contextual Retrieval)

While RAG excels at retrieving information from embedded document collections, many real-world scenarios require a different approach. When information is stored in structured databases, file systems, or complex data structures, *exploration* is often more effective than semantic search. This is where the Navigator pattern shines.

4.2.1 Retrieving Information based on Data Structure and Model Intelligence

The Navigator pattern, also called the Retriever pattern, differs from RAG in a fundamental way:

- **RAG** relies on *semantic information*: It embeds text and finds similar meanings
- **Navigator** relies on *structural information*: It understands how data is organized and explores accordingly

When Navigator Outperforms RAG
Consider these scenarios:
Scenario 1: Codebase Navigation
You're building an agent that answers questions about a large codebase. RAG could embed code files and retrieve relevant ones. However, code has rich structure:

- Files are organized in directories
- Functions call other functions
- Classes inherit from other classes
- Imports create dependency graphs

A Navigator agent can:

1. Start at a high-level directory
2. Read file names to identify relevant modules
3. Open specific files to examine their structure
4. Follow import statements to related code
5. Trace function calls to understand data flow

This exploration-based approach leverages the codebase's inherent structure in ways that semantic embedding cannot.
Scenario 2: Database Query
You need to answer questions about data in a SQL database. RAG would require embedding database contents (expensive and impractical for large databases). Instead, a Navigator agent can:

1. Examine the database schema
2. Construct appropriate SQL queries
3. Execute queries and analyze results
4. Refine queries based on initial results

The agent navigates the database structure intelligently rather than relying on pre-embedded content.

Scenario 3: File System Exploration

An agent helping users find files in a large organizational file system can:

1. List directory contents
2. Navigate to relevant subdirectories based on naming
3. Search file contents when needed
4. Follow symlinks or shortcuts

The Core Principle

The Navigator pattern works because:

1. **Data has structure**: File systems have directories, databases have schemas, codebases have modules
2. **Structure provides clues**: Directory names, table names, file names contain semantic information
3. **Agents can reason about structure**: Language models can understand "if I'm looking for authentication code, I should check the /auth directory"
4. **Exploration is efficient**: Following structure is often faster than searching everything

RAG vs. Navigator: When to Use Each

Characteristic	RAG	Navigator
Data organization	Unstructured/semi-structured	Highly structured
Data size	Moderate (embeddable)	Arbitrarily large
Query style	Semantic similarity	Structural exploration
Primary tool	Embedding model	Exploration tools
Example use cases	Documentation, articles	Databases, codebases

Often, the best approach combines both: use Navigator to find relevant regions, then use RAG within those regions for semantic search. Alternatively, the process can work in reverse: use RAG to identify semantically relevant sections of a large dataset, then employ Navigator to explore the structure and relationships within those sections.

4.2.2 Utilizing Specialized Exploration Tools (Terminal, Database Access)

The Navigator pattern is implemented by providing agents with exploration tools tailored to the data structure they're navigating.

File System Navigation Tools

For navigating file systems:

```python
## tools.py
from openai import function_tool
import os
from pathlib import Path

@function_tool
def list_directory(path: str) -> list[str]:
    """
    List contents of a directory.

    Args:
        path: Directory path to list

    Returns:
        List of files and subdirectories
    """
    try:
        path_obj = Path(path)
        if not path_obj.exists():
            return [f"Error: Path does not exist: {path}"]

        if not path_obj.is_dir():
            return [f"Error: Not a directory: {path}"]

        contents = os.listdir(path)

        # Annotate directories vs files
        annotated = []
        for item in contents:
            item_path = path_obj / item
            if item_path.is_dir():
                annotated.append(f"[DIR] {item}")
            else:
                annotated.append(f"[FILE] {item}")

        return annotated
    except Exception as e:
        return [f"Error listing directory: {str(e)}"]

@function_tool
def read_file(path: str, max_lines: int = 100) -> str:
    """
    Read contents of a file.

    Args:
        path: File path to read
        max_lines: Maximum number of lines to read

    Returns:
        File contents (or error message)
    """
    try:
        with open(path, 'r', encoding='utf-8') as f:
            lines = []
            for i, line in enumerate(f):
```

```
                if i >= max_lines:
                    lines.append(f"\n... (file continues, {max_lines} line limit)")
                    break
                lines.append(line.rstrip())

            return "\n".join(lines)
    except Exception as e:
        return f"Error reading file: {str(e)}"

@function_tool
def search_files(directory: str, pattern: str) -> list[str]:
    """
    Search for files matching a pattern.

    Args:
        directory: Directory to search in
        pattern: Glob pattern (e.g., "*.py", "test_*")

    Returns:
        List of matching file paths
    """
    try:
        path_obj = Path(directory)
        matches = list(path_obj.glob(f"**/{pattern}"))
        return [str(m.relative_to(path_obj)) for m in matches[:50]]
    except Exception as e:
        return [f"Error searching: {str(e)}"]
```

Database Navigation Tools

For navigating SQL databases:

```
import sqlite3
from typing import list, dict

@function_tool
def list_tables(db_path: str) -> list[str]:
    """
    List all tables in the database.

    Args:
        db_path: Path to SQLite database

    Returns:
        List of table names
    """
    try:
        conn = sqlite3.connect(db_path)
        cursor = conn.cursor()

        cursor.execute(
```

```
            "SELECT name FROM sqlite_master WHERE type='table'"
        )
        tables = [row[0] for row in cursor.fetchall()]

        conn.close()
        return tables
    except Exception as e:
        return [f"Error: {str(e)}"]

@function_tool
def describe_table(db_path: str, table_name: str) -> list[dict]:
    """
    Get schema information for a table.

    Args:
        db_path: Path to SQLite database
        table_name: Name of table to describe

    Returns:
        List of column information dicts
    """
    try:
        conn = sqlite3.connect(db_path)
        cursor = conn.cursor()

        cursor.execute(f"PRAGMA table_info({table_name})")
        columns = []
        for row in cursor.fetchall():
            columns.append({
                "name": row[1],
                "type": row[2],
                "nullable": not row[3],
                "primary_key": bool(row[5])
            })

        conn.close()
        return columns
    except Exception as e:
        return [{"error": str(e)}]

@function_tool
def execute_query(db_path: str, query: str, limit: int = 100) -> list[dict]:
    """
    Execute a SELECT query on the database.

    Args:
        db_path: Path to SQLite database
        query: SQL SELECT query
        limit: Maximum rows to return

    Returns:
        Query results as list of dicts
    """
```

```
    try:
        # Security: Only allow SELECT queries
        if not query.strip().upper().startswith("SELECT"):
            return [{"error": "Only SELECT queries are allowed"}]

        conn = sqlite3.connect(db_path)
        conn.row_factory = sqlite3.Row  # Return rows as dicts
        cursor = conn.cursor()

        cursor.execute(query)
        rows = cursor.fetchmany(limit)

        results = [dict(row) for row in rows]

        conn.close()
        return results
    except Exception as e:
        return [{"error": str(e)}]
```

Code Navigation Tools

For navigating codebases:

```
import ast

@function_tool
def list_functions_in_file(file_path: str) -> list[dict]:
    """
    List all functions defined in a Python file.

    Args:
        file_path: Path to Python file

    Returns:
        List of function information dicts
    """
    try:
        with open(file_path, 'r') as f:
            tree = ast.parse(f.read())

        functions = []
        for node in ast.walk(tree):
            if isinstance(node, ast.FunctionDef):
                # Extract docstring if present
                docstring = ast.get_docstring(node)

                functions.append({
                    "name": node.name,
                    "line": node.lineno,
                    "args": [arg.arg for arg in node.args.args],
                    "docstring": docstring or "No docstring
```

```
    "
                })

        return functions
    except Exception as e:
        return [{"error": str(e)}]

@function_tool
def find_function_calls(file_path: str, function_name: str)
    -> list[int]:
    """
    Find all locations where a function is called.

    Args:
        file_path: Path to Python file
        function_name: Name of function to find calls for

    Returns:
        List of line numbers where function is called
    """
    try:
        with open(file_path, 'r') as f:
            tree = ast.parse(f.read())

        call_lines = []
        for node in ast.walk(tree):
            if isinstance(node, ast.Call):
                if isinstance(node.func, ast.Name):
                    if node.func.id == function_name:
                        call_lines.append(node.lineno)

        return call_lines
    except Exception as e:
        return [f"Error: {str(e)}"]
```

Building a Navigator Agent

A Navigator agent combines these tools with instructions that encourage exploration:

```
## agents/code_navigator.py
from openai import Agent
from ..tools import (
    list_directory, read_file, search_files,
    list_functions_in_file, find_function_calls
)

instructions = """You are a code navigation specialist.
   Your task is to
explore codebases to find relevant information.

Your approach:
1. Start by getting an overview of the directory structure
2. Identify relevant modules based on naming and
```

```
    organization
3. Examine specific files to understand their contents
4. Follow references (imports, function calls) to related
    code
5. Synthesize findings into a clear explanation

Tools at your disposal:
- list_directory: See what's in a directory
- read_file: Read file contents
- search_files: Find files matching patterns
- list_functions_in_file: See functions in a Python file
- find_function_calls: Track where functions are used

Take a methodical, exploratory approach. Don't try to read
    everything at
once; navigate intelligently based on what you find.
"""

code_navigator = Agent(
    name="code_navigator",
    model="gpt-5",  # Needs strong reasoning for navigation
    instructions=instructions,
    tools=[
        list_directory,
        read_file,
        search_files,
        list_functions_in_file,
        find_function_calls
    ]
)
```

Example: Using the Navigator

```
## Ask the navigator to find authentication code
query = """In this codebase, find where user authentication
    is implemented.
I need to understand how login credentials are verified."""

response = code_navigator.run(query)

# The agent will:
# 1. List the root directory to see structure
# 2. Identify likely directories (e.g., /auth, /users, /
    security)
# 3. Explore those directories
# 4. Read relevant files (e.g., auth.py, login.py)
# 5. List functions to find authentication-related ones
# 6. Provide a comprehensive answer about the auth
    implementation
```

The Navigator agent doesn't just search for keywords; it intelligently explores the codebase structure, following the same intuitions a human developer would use.

Navigator Best Practices

1. Bounded Exploration

Prevent infinite exploration with limits:

```
## In agent instructions:
"""You have a budget of 20 tool calls. Use them wisely.
   Start with high-level
exploration, then dive deep only into the most promising
   areas."""
```

2. Contextual Memory

Track what's been explored to avoid redundant checks:

```
## Add to agent instructions:
"""Keep track of what you've already examined. Don't re-
   read the same files
or directories repeatedly."""
```

3. Incremental Results

For long-running explorations, return intermediate findings:

```
"""As you explore, provide incremental updates about what
   you're finding.
Don't wait until you've explored everything to give your
   first response."""
```

4. Safety Constraints

Limit what the Navigator can access:

- Restrict to specific directories (no system files)
- Read-only access (no modification tools)
- Query limits for databases
- Sandboxed execution environments

4.3 Code Interpreter

The third universal pattern, Code Interpreter, is perhaps the most powerful: it gives agents the ability to write and execute arbitrary code. This capability, first popularized by OpenAI's Code Interpreter feature, transforms agents from passive reasoners into active problem-solvers.

4.3.1 Providing Unlimited Problem-Solving Ability through Arbitrary Code Execution

Language models excel at many tasks, but they have fundamental limitations:

- **Arithmetic**: Models approximate arithmetic rather than computing it precisely
- **Complex calculations**: Statistical analysis, numerical optimization, simulation
- **Data processing**: Transforming large datasets, parsing complex formats
- **Visualization**: Creating charts, graphs, diagrams
- **Specialized algorithms**: Implementing algorithms not representable in natural language

Code Interpreter solves these limitations by allowing the agent to write Python code (or another language), execute it in a sandboxed environment, and incorporate the results into its reasoning.

The Power of Code Execution

Consider this query: "I have sales data for the last year. Calculate the month-over-month growth rate, identify seasonal patterns, and create a visualization."

Without Code Interpreter, the agent would struggle:

- It can't accurately compute growth rates
- It can't perform statistical analysis for pattern detection
- It can't create actual visualizations

With Code Interpreter:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

# Load data
data = pd.read_csv('sales_data.csv')

# Calculate month-over-month growth
data['growth_rate'] = data['sales'].pct_change() * 100

# Seasonal decomposition
from statsmodels.tsa.seasonal import seasonal_decompose
decomposition = seasonal_decompose(data['sales'], model='
    additive', period=12)

# Visualization
fig, axes = plt.subplots(3, 1, figsize=(12, 10))
data['sales'].plot(ax=axes[0], title='Monthly Sales')
data['growth_rate'].plot(ax=axes[1], title='Month-over-
    Month Growth Rate')
decomposition.seasonal.plot(ax=axes[2], title='Seasonal
    Pattern')
```

```
plt.savefig('sales_analysis.png')
```

The agent writes this code, executes it, examines the results and visualization, and provides insights based on the actual computed data.

Why Code Interpreter is "Unlimited"

Given sufficient time and a capable environment, code can solve almost any computational problem. Code Interpreter gives agents:

- **Precision**: Exact calculations instead of approximations
- **Scalability**: Process datasets of any size
- **Flexibility**: Solve problems not anticipated during agent design
- **Tool Creation**: Generate custom tools on-the-fly for specific needs

This last point is crucial: with Code Interpreter, you don't need to anticipate every tool an agent might need. The agent can create tools as required.

Real-World Example: OpenAI's Deep Research

OpenAI's Deep Research agent uses Code Interpreter despite being primarily a research tool. Why? Because research involves:

- Numerical analysis of data in papers
- Statistical validation of claims
- Creating visualizations to understand trends
- Processing and comparing datasets from multiple sources

Without Code Interpreter, the agent would need separate tools for each type of calculation. With it, the agent handles unexpected computational needs naturally.

4.3.2 Handling Calculations, Visuals, and Unplanned Tools

Let's explore specific use cases where Code Interpreter excels.

Use Case 1: Complex Calculations

Problem: Calculate the compound annual growth rate (CAGR) and perform sensitivity analysis.

Without Code Interpreter:

- Need a dedicated CAGR calculator tool
- Need a sensitivity analysis tool
- Tools must be written and maintained
- Limited flexibility for variations

With Code Interpreter:

```
def calculate_cagr(beginning_value, ending_value, num_years):
    """Calculate Compound Annual Growth Rate."""
    return ((ending_value / beginning_value) ** (1 / num_years) - 1) * 100

def sensitivity_analysis(base_value, variations):
    """Perform sensitivity analysis."""
    results = {}
    for factor, change_pct in variations.items():
        modified = base_value * (1 + change_pct / 100)
        results[factor] = {
            'value': modified,
            'change': modified - base_value
        }
    return results

# Example usage
beginning = 100000
ending = 150000
years = 5

cagr = calculate_cagr(beginning, ending, years)
print(f"CAGR: {cagr:.2f}%")

# Sensitivity
variations = {
    'optimistic': 10,
    'pessimistic': -10,
    'base': 0
}
sensitivity = sensitivity_analysis(ending, variations)
for scenario, data in sensitivity.items():
    print(f"{scenario}: ${data['value']:.2f}")
```

Use Case 2: Data Visualization

Problem: Create a custom visualization for multi-dimensional data.

```
import matplotlib.pyplot as plt
import numpy as np

# Sample data
categories = ['Q1', 'Q2', 'Q3', 'Q4']
actual = [120, 145, 165, 180]
budget = [130, 140, 160, 175]
forecast = [125, 150, 170, 185]

x = np.arange(len(categories))
width = 0.25

fig, ax = plt.subplots(figsize=(10, 6))
ax.bar(x - width, actual, width, label='Actual', color='#2
```

```
    E86AB')
ax.bar(x, budget, width, label='Budget', color='#A23B72')
ax.bar(x + width, forecast, width, label='Forecast', color=
    '#F18F01')

ax.set_xlabel('Quarter')
ax.set_ylabel('Revenue ($K)')
ax.set_title('Quarterly Revenue Comparison')
ax.set_xticks(x)
ax.set_xticklabels(categories)
ax.legend()
ax.grid(axis='y', alpha=0.3)

plt.tight_layout()
plt.savefig('revenue_comparison.png', dpi=300)
plt.show()
```

The agent creates precisely the visualization needed, without requiring a pre-built chart tool.

Use Case 3: Data Transformation

Problem: Parse and transform data from an unusual format.

```
import re
import json

def parse_custom_format(text):
    """Parse a custom data format into structured JSON."""
    # Example: "User:John|Age:30|City:NYC;User:Jane|Age:25|
    City:LA"

    records = []
    for record_str in text.split(';'):
        record = {}
        for field in record_str.split('|'):
            if ':' in field:
                key, value = field.split(':', 1)
                # Try to convert to number if possible
                try:
                    value = int(value)
                except ValueError:
                    try:
                        value = float(value)
                    except ValueError:
                        pass  # Keep as string
                record[key.strip()] = value
        if record:
            records.append(record)

    return records

# Parse the data
raw_data = "User:John|Age:30|City:NYC;User:Jane|Age:25|City
    :LA"
```

```
parsed = parse_custom_format(raw_data)

# Convert to JSON
print(json.dumps(parsed, indent=2))
```

Implementing Code Interpreter

Modern agent frameworks often provide built-in code execution capabilities. For OpenAI's Agents SDK:

```
from openai import Agent

code_interpreter_agent = Agent(
    name="analyst",
    model="gpt-5",
    instructions="""You are a data analyst. Use code execution to perform
    precise calculations, create visualizations, and process data. Always
    use code for numerical operations rather than approximating.""",
    tools=["code_interpreter"]  # Built-in tool
)
```

For custom implementations, you need a safe execution environment:

```
from openai import function_tool
import subprocess
import tempfile
import os

@function_tool
def execute_python_code(code: str) -> dict:
    """
    Execute Python code in a sandboxed environment.

    Args:
        code: Python code to execute

    Returns:
        Dict with stdout, stderr, and success status
    """
    try:
        # Create temporary file
        with tempfile.NamedTemporaryFile(
            mode='w',
            suffix='.py',
            delete=False
        ) as f:
            f.write(code)
            temp_file = f.name
```

```
26
27          # Execute with timeout and resource limits
28          result = subprocess.run(
29              ['python', temp_file],
30              capture_output=True,
31              text=True,
32              timeout=30,  # 30-second timeout
33              # Note: In production, use proper sandboxing (
        Docker, etc.)
34          )
35
36          # Clean up
37          os.unlink(temp_file)
38
39          return {
40              'success': result.returncode == 0,
41              'stdout': result.stdout,
42              'stderr': result.stderr,
43              'exit_code': result.returncode
44          }
45
46      except subprocess.TimeoutExpired:
47          os.unlink(temp_file)
48          return {
49              'success': False,
50              'stdout': '',
51              'stderr': 'Execution timed out (>30s)',
52              'exit_code': -1
53          }
54      except Exception as e:
55          return {
56              'success': False,
57              'stdout': '',
58              'stderr': str(e),
59              'exit_code': -1
60          }
```

Security Considerations
Code execution is powerful but dangerous. Essential security measures:

- **Sandboxing**: Execute code in isolated environments (Docker containers, VMs)
- **Resource limits**: CPU time, memory, disk space, network access
- **Timeouts**: Prevent infinite loops
- **File system isolation**: Restrict file access to specific directories
- **Network isolation**: Block or restrict network access
- **Package restrictions**: Limit available libraries to safe subset

For production systems, use dedicated code execution services (e.g., AWS Lambda with strict IAM, Google Cloud Functions, or specialized sandboxing solutions like Pyodide for browser-based execution).

Code Interpreter Best Practices

1. Encourage Code Use in Instructions

```
"""For any numerical calculations, statistical analysis, or
    data processing:
- ALWAYS use code execution
- Don't approximate calculations in text
- Show your code so results are reproducible
"""
```

2. Provide Common Libraries

Pre-install useful libraries in your execution environment:

- Data: pandas, numpy
- Visualization: matplotlib, seaborn, plotly
- Statistics: scipy, statsmodels
- ML: scikit-learn (if appropriate)

3. Handle Errors Gracefully

```
"""If your code produces an error:
1. Read the error message carefully
2. Identify the issue
3. Write corrected code
4. You can iterate up to 3 times to fix code errors
"""
```

4. Return Results Clearly

```
"""After executing code:
1. Explain what the code does
2. Present the results clearly
3. If you created a visualization, describe it
4. Provide interpretation of the findings
"""
```

4.4 The Tool Selector Pattern

The final universal pattern addresses a scaling challenge: what happens when your agent needs access to hundreds or thousands of tools?

4.4.1 Necessity: When Tool Inventory Exceeds Context Window Capacity

Every tool provided to an agent consumes context window space. The tool's name, description, parameters, and examples all occupy tokens. For agents with access to many tools, this creates a problem:

Example: Zapier-like Automation Agent

Zapier integrates with over 5,000 applications. Imagine building a Zapier agent that can automate workflows across these applications. Each application might expose 5-20 actions (tools), resulting in:

$$5,000 \text{ apps} \times 10 \text{ actions/app} = 50,000 \text{ potential tools}$$

Even with concise descriptions, if each tool takes 50 tokens, that's:

$$50,000 \text{ tools} \times 50 \text{ tokens/tool} = 2,500,000 \text{ tokens}$$

This exceeds the context window of any current model. Even if we could fit all tools, providing that many options degrades the model's ability to select the right one.

The Tool Selection Problem

The challenge is: how do you give an agent access to a large tool inventory without overwhelming its context window or decision-making ability?

The Tool Selector pattern solves this through a two-stage process:

1. **Selection stage**: Choose the most relevant tools from the full inventory
2. **Execution stage**: Provide only selected tools to the agent for actual use

4.4.2 Implementation: Combining RAG or Navigator for Tool Selection

The Tool Selector pattern leverages the RAG and Navigator patterns we've already explored, applying them to tool selection rather than information retrieval.

Approach 1: RAG-Based Tool Selection

Treat tool descriptions as documents and use semantic search to find relevant tools:

Step 1: Create Tool Embeddings

```
from typing import NamedTuple

class ToolMetadata(NamedTuple):
    name: str
    description: str
    category: str
    parameters: dict

def create_tool_embedding_store(tools: list[ToolMetadata])
    -> SimpleRAGStore:
    """
    Create an embedding store for tool selection.
```

```
    Args:
        tools: List of tool metadata

    Returns:
        RAG store with embedded tool descriptions
    """
    store = SimpleRAGStore()

    for tool in tools:
        # Create rich description for embedding
        tool_text = f"""
        Tool: {tool.name}
        Category: {tool.category}
        Description: {tool.description}
        Parameters: {', '.join(tool.parameters.keys())}
        """

        # Store with metadata
        store.chunks.append(tool_text)
        store.metadata.append({
            'tool_name': tool.name,
            'category': tool.category,
            'tool_object': tool
        })

    # Embed all tool descriptions
    store.embeddings = embed_chunks(store.chunks)

    return store
```

Step 2: Select Tools Based on Query

```
def select_relevant_tools(
    user_query: str,
    tool_store: SimpleRAGStore,
    max_tools: int = 20
) -> list[ToolMetadata]:
    """
    Select the most relevant tools for a query.

    Args:
        user_query: The user's request
        tool_store: RAG store of tool embeddings
        max_tools: Maximum tools to select

    Returns:
        List of selected tool metadata
    """
    # Retrieve most relevant tool descriptions
    results = tool_store.retrieve(user_query, top_k=
    max_tools)

    # Extract tool objects
```

```
    tools = [r['metadata']['tool_object'] for r in results]

    return tools
```

Step 3: Provide Selected Tools to Agent

```
from openai import Agent

def create_agent_with_selected_tools(
    user_query: str,
    tool_store: SimpleRAGStore,
    base_instructions: str
) -> Agent:
    """
    Create an agent with dynamically selected tools.

    Args:
        user_query: User's request (for tool selection)
        tool_store: Tool embedding store
        base_instructions: Agent's base instructions

    Returns:
        Agent configured with relevant tools
    """
    # Select relevant tools
    relevant_tools = select_relevant_tools(user_query,
    tool_store, max_tools=15)

    # Convert to actual tool functions (implementation
    specific)
    tool_functions = [tool.to_function() for tool in
    relevant_tools]

    # Create agent with selected tools
    agent = Agent(
        name="task_agent",
        model="gpt-5",
        instructions=base_instructions,
        tools=tool_functions
    )

    return agent
```

Complete Example: Zapier-Style Agent

```
## Define tool inventory (abbreviated)
tools = [
    ToolMetadata(
        name="gmail_send_email",
        description="Send an email via Gmail",
        category="gmail",
```

```
        parameters={"to": "str", "subject": "str", "body": "str"}
    ),
    ToolMetadata(
        name="slack_post_message",
        description="Post a message to a Slack channel",
        category="slack",
        parameters={"channel": "str", "text": "str"}
    ),
    ToolMetadata(
        name="google_calendar_create_event",
        description="Create an event in Google Calendar",
        category="google_calendar",
        parameters={"title": "str", "start": "datetime", "end": "datetime"}
    ),
    # ... 49,997 more tools ...
]

# Create tool embedding store
tool_store = create_tool_embedding_store(tools)
tool_store.save("tool_embeddings.pkl")

# User request
user_query = "Send a summary email to my team and post it in Slack"

# Select relevant tools (will retrieve gmail and slack tools)
agent = create_agent_with_selected_tools(
    user_query,
    tool_store,
    base_instructions="You are an automation assistant..."
)

# Agent now has access to ~15 relevant tools instead of 50,000
response = agent.run(user_query)
```

Approach 2: Navigator-Based Tool Selection

For tools organized hierarchically (e.g., by application, then by action type), use a Navigator agent to explore the tool taxonomy:

```
## Tool hierarchy
tools_hierarchy = {
    "gmail": {
        "email": ["send_email", "search_emails", "delete_email"],
        "labels": ["create_label", "apply_label"],
    },
    "slack": {
        "messages": ["post_message", "update_message", "delete_message"],
```

```
            "channels": ["create_channel", "list_channels"],
        },
        # ... more apps ...
}

@function_tool
def list_tool_categories() -> list[str]:
    """List available tool categories (apps)."""
    return list(tools_hierarchy.keys())

@function_tool
def list_tool_subcategories(category: str) -> list[str]:
    """List subcategories within a category."""
    return list(tools_hierarchy.get(category, {}).keys())

@function_tool
def list_tools_in_subcategory(category: str, subcategory: str) -> list[str]:
    """List specific tools in a subcategory."""
    return tools_hierarchy.get(category, {}).get(subcategory, [])

# Navigator agent explores hierarchy to find relevant tools
tool_selector_agent = Agent(
    name="tool_selector",
    model="gpt-5",
    instructions="""You help select relevant tools. Given a user request,
    explore the tool hierarchy to identify which specific tools would be
    useful. Return a list of tool names.""",
    tools=[list_tool_categories, list_tool_subcategories, list_tools_in_subcategory]
)

# User query
query = "Send email and post to Slack"

# Selector explores: categories -> finds "gmail", "slack" ->
# explores subcategories -> finds "email", "messages" -> returns specific tools
selected_tool_names = tool_selector_agent.run(query)

# Load those specific tools for the main agent
# ... (implementation specific)
```

Approach 3: Hybrid (RAG + Navigator)
The most robust approach combines both:

1. Use RAG to narrow from 50,000 tools to 200 candidates
2. Use Navigator to explore the structure of those 200 tools
3. Select the final 15-20 most relevant tools

```python
def hybrid_tool_selection(
    user_query: str,
    tool_store: SimpleRAGStore,
    tool_hierarchy: dict
) -> list[str]:
    """
    Select tools using hybrid RAG + Navigator approach.

    Args:
        user_query: User's request
        tool_store: RAG-based tool store
        tool_hierarchy: Hierarchical tool organization

    Returns:
        List of selected tool names
    """
    # Stage 1: RAG to get ~200 candidates
    candidates = select_relevant_tools(user_query, tool_store, max_tools=200)

    # Stage 2: Build Navigator with candidates
    candidate_hierarchy = build_hierarchy_from_tools(candidates)

    # Stage 3: Navigator explores and selects final tools
    navigator = create_tool_navigator(candidate_hierarchy)
    final_tools = navigator.run(f"Select best tools for: {user_query}")

    return final_tools
```

Tool Selector Best Practices

1. Rich Tool Descriptions

Quality of tool selection depends on description quality:

```python
## Poor description
"Send email"

# Good description
"Send an email via Gmail. Use this when the user wants to compose and
send email to recipients. Supports HTML formatting and attachments."
```

2. Tool Categorization

Organize tools into logical categories to improve both RAG and Navigator performance:

- By application (Gmail, Slack, etc.)
- By action type (create, read, update, delete)

- By domain (communication, storage, analytics)

3. Caching
Cache tool selections for common query patterns:

```
## If multiple users ask similar questions, reuse tool selections
cache = {
    "send email": ["gmail_send_email", "outlook_send_email"],
    "schedule meeting": ["google_calendar_create", "outlook_calendar_create"]
}
```

4. Monitoring and Refinement
Track which tools are selected and used:

```
## Log tool selection quality
def log_tool_usage(query: str, selected_tools: list, actually_used: list):
    """Track if selected tools were actually useful."""
    precision = len(set(actually_used) & set(selected_tools)) / len(selected_tools)
    recall = len(set(actually_used) & set(selected_tools)) / len(actually_used)

    # Log for analysis
    logger.info(f"Query: {query}, Precision: {precision:.2f}, Recall: {recall:.2f}")
```

Use this data to refine tool descriptions and selection strategies.

This chapter introduced four universal reusable agent patterns that extend agent capabilities beyond their inherent limitations:

- **RAG (Retrieval Augmented Generation)**: Overcomes knowledge boundaries by retrieving relevant information from embedded document collections. Reduces hallucination and enables agents to work with domain-specific knowledge. Implementation involves chunking documents, embedding them, and retrieving via semantic similarity.

- **Navigator Pattern**: Enables intelligent exploration of structured data (codebases, databases, file systems) by providing specialized exploration tools. Unlike RAG's semantic approach, Navigator leverages data structure and model intelligence to navigate effectively.

- **Code Interpreter**: Provides unlimited problem-solving capability through arbitrary code execution. Enables precise calculations, data visualization, complex transformations, and on-the-fly tool creation. Requires careful sandboxing and security measures.

- **Tool Selector**: Manages large tool inventories that exceed context window capacity. Uses RAG, Navigator, or hybrid approaches to select relevant tools dynamically, enabling agents to work with thousands of potential tools while maintaining focus.

These patterns are universal because they address fundamental agent limitations across all domains. They are reusable because, once implemented, they can be applied to multiple agents and services. Most importantly, they almost always improve system quality—incorporating these patterns into your architecture is rarely harmful and often transformative.

In the MUTTA architecture:

- RAG knowledge bases are typically initialized in the manager
- Navigator tools are defined in `tools.py`
- Code Interpreter is provided as a built-in or custom tool
- Tool Selector operates at the manager level to configure agents dynamically

With these patterns in our toolkit, we can build agents that access vast knowledge bases, explore complex data structures, perform precise computations, and select from enormous tool inventories. However, even with these capabilities, agents still make errors. In Chapter 5, we explore systematic approaches to detecting, correcting, and preventing errors in agent systems, ensuring that our powerful patterns operate reliably in production environments.

Chapter 5

Implementing Robust Error Correction Systems

In previous chapters, we established the foundations of agent systems (Chapter 1), introduced the MUTTA architecture (Chapter 2), developed techniques for ensuring input quality (Chapter 3), and explored reusable patterns that extend agent capabilities (Chapter 4). These elements provide a strong foundation for building sophisticated agent systems.

However, even with optimal architecture, high-quality inputs, and powerful patterns like RAG and Code Interpreter, agent systems will inevitably produce errors. Language models hallucinate, tools fail, edge cases emerge, and complex multi-step reasoning goes astray. The question is not whether errors will occur, but how we detect, correct, and prevent them.

This chapter introduces systematic approaches to error correction—patterns and techniques for ensuring that agent outputs are accurate, grounded, and reliable. We explore four complementary strategies:

1. **Grounding principles**: Understanding what it means for an agent's output to be "correct" and how to verify correctness

2. **Knowledge-based correction**: Using RAG and Navigator patterns to verify factual claims

3. **Strategic verification**: Choosing when to apply error correction based on action criticality

4. **Logical validation**: Verifying logical soundness for mathematical and reasoning tasks

Error correction is not a single technique but a toolkit of complementary approaches. The key is selecting the right correction strategy for each situation, balancing thoroughness against cost and latency. By the end of this chapter, you will understand how to design error correction systems that match your application's reliability requirements.

5.1 Principles of Grounding and Verification

Before diving into specific error correction techniques, we must establish what we mean by "correctness" and "grounding" in the context of agent systems.

5.1.1 Error Correction through Fact Verification or Logical Soundness

An agent's output can be incorrect in two fundamentally different ways, each requiring different correction approaches.

Type 1: Factual Errors

Factual errors occur when an agent makes claims that contradict verifiable information. Examples:

- "The capital of France is Lyon" (factually incorrect)
- "Python 3.12 was released in 2025" (wrong date)
- "The boiling point of water is 90°C at sea level" (wrong measurement)
- "This gene codes for insulin" (when it actually codes for a different protein)

Factual errors can be detected and corrected through **fact verification**—comparing the agent's claims against authoritative sources (documents, databases, APIs).

Type 2: Logical Errors

Logical errors occur when reasoning is internally inconsistent or violates logical rules, regardless of factual content. Examples:

- "All birds can fly. Penguins are birds. Therefore, penguins cannot fly." (logical inconsistency)
- "If $x > 5$ and $x < 3$, then $x = 4$" (mathematical impossibility)
- "The code will reach line 50" (when line 45 contains a return statement)
- A proof that assumes what it's trying to prove (circular reasoning)

Logical errors require **logical soundness checking**—verifying that reasoning steps are valid, conclusions follow from premises, and no contradictions exist.

The Grounding Concept

Grounding means anchoring an agent's outputs to something verifiable:

- **Fact-grounded**: Claims are backed by retrievable documents, database records, or API responses
- **Logic-grounded**: Reasoning steps are validated as logically sound, potentially using formal verification tools

An ungrounded agent output might sound plausible but lacks verification. A grounded output can be traced back to its source of truth.

Example: Grounded vs. Ungrounded Responses

User query: "What are the side effects of Medication X?"

Ungrounded response:

```
"Medication X commonly causes headaches, dizziness, and nausea.
Rare side effects include liver damage. Consult your doctor before use."
```

This might be accurate, but we cannot verify it. The agent could be hallucinating.

Grounded response:

```
"According to the FDA-approved prescribing information (retrieved
from drugs.com, last updated 2024-03-15):

Common side effects (>10% of patients):
- Headache (15%)
- Dizziness (12%)

Rare but serious side effects (<1%):
- Elevated liver enzymes (monitor liver function)

Source: [Link to prescribing information]"
```

This response is grounded—we can verify the claims by checking the cited source.

When Grounding Matters Most

Not all agent outputs require the same level of grounding. Consider:

High grounding requirement:

- Medical advice
- Legal guidance
- Financial recommendations
- Scientific claims
- Safety-critical decisions

Lower grounding requirement:

- Creative writing
- Brainstorming ideas
- General conversation
- Subjective opinions

The error correction strategies we develop should be applied most rigorously where grounding is critical.

The Error Correction Decision

For each agent output, we can ask:

1. **Can we verify this?** Is there a source of truth (documents, databases, logical rules)?
2. **Should we verify this?** Does the application require high confidence in correctness?
3. **How should we verify?** Fact-checking, logic-checking, or both?
4. **When should we verify?** Before committing to actions, periodically, or on every step?

The remainder of this chapter provides techniques for implementing verification systematically.

The Cost-Reliability Trade-off

Error correction is not free. Each verification step adds:

- **Latency**: Additional model calls or computations
- **Cost**: More API calls, more compute resources
- **Complexity**: More code to maintain

The art of error correction is achieving sufficient reliability without excessive overhead. We'll explore strategies for both lightweight continuous checking and heavyweight critical verification.

5.2 Knowledge-Based Correction Techniques

The first category of error correction techniques leverages the RAG and Navigator patterns from Chapter 4 to verify factual claims. If an agent can retrieve information to augment its responses, it can also retrieve information to verify those responses.

5.2.1 The Lazy Agent Check (RAG/Navigator Based)

The **Lazy Agent Check** is a verification agent that checks the outputs of other agents against a knowledge base. It's called "lazy" because it operates asynchronously or periodically rather than on every step, minimizing performance impact.

Core Concept

The Lazy Agent Check pattern works as follows:

1. A primary agent produces an output
2. A verification agent receives: (a) the input context, (b) the output, and (c) access to verification tools
3. The verification agent checks the output's claims against the knowledge base
4. If errors are detected, the system either halts or provides feedback for correction

Implementation Approaches

There are two architectural approaches, each with different trade-offs.

Parallel vs. Series Implementation and Production Halting

Series Implementation

In series mode, the verification agent runs *before* the output is finalized:

```python
def run_with_series_verification(self, user_input: str) -> str:
    """Run agent with series verification."""

    # Step 1: Primary agent generates output
    primary_output = self.primary_agent.run(user_input)

    # Step 2: Verification agent checks output
    verification_result = self.verification_agent.run(
        f"""Verify the following output:

        Input: {user_input}
        Output: {primary_output}

        Check all factual claims. Report any errors found."""
    )

    # Step 3: Handle verification results
    if verification_result.errors_found:
        # Option A: Return error to user
        return f"Error in processing: {verification_result.error_details}"

        # Option B: Request correction from primary agent
        corrected = self.primary_agent.run(
            f"""Your previous output had errors:
            {verification_result.error_details}

            Please provide a corrected response to: {user_input}"""
        )
        return corrected

    # No errors found, return original output
    return primary_output
```

Advantages of series mode:

- Errors are caught before reaching the user
- Can implement correction loops
- Simpler reasoning about system state

Disadvantages:

- Adds latency to every request
- Doubles the minimum response time

- More expensive (two agent calls per request)

Parallel Implementation

In parallel mode, the verification agent runs concurrently with downstream processing:

```python
import asyncio

async def run_with_parallel_verification(self, user_input: str) -> str:
    """Run agent with parallel verification."""

    # Step 1: Primary agent generates output
    primary_output = await self.primary_agent.run_async(user_input)

    # Step 2: Start verification and downstream processing in parallel
    verification_task = asyncio.create_task(
        self.verification_agent.run_async(
            f"Verify: {primary_output}"
        )
    )

    downstream_task = asyncio.create_task(
        self.downstream_processing(primary_output)
    )

    # Step 3: Check if verification completes with errors
    # before downstream finishes
    verification_result = await verification_task

    if verification_result.errors_found:
        # Cancel downstream if still running
        if not downstream_task.done():
            downstream_task.cancel()

        # Handle error (halt production)
        return f"Error detected: {verification_result.error_details}"

    # Wait for downstream to complete
    result = await downstream_task
    return result
```

Advantages of parallel mode:

- Lower latency if verification finds no errors
- Better resource utilization
- Scales better under load

Disadvantages:

- More complex to implement
- Wasted computation if errors are found
- Requires async support throughout the system

Production Halting

A critical feature of the Lazy Agent Check is **production halting**—stopping the pipeline when errors are detected.

```python
class VerificationResult(BaseModel):
    """Result of verification check."""
    errors_found: bool
    error_details: list[str]
    error_step: Optional[str] = None  # Which step had the error
    severity: str = "medium"  # low, medium, high, critical

def halt_if_critical(verification: VerificationResult) -> None:
    """Halt production if critical errors found."""
    if verification.errors_found and verification.severity in ["high", "critical"]:
        raise ProductionHaltException(
            f"Critical error detected at {verification.error_step}: "
            f"{verification.error_details}"
        )
```

Production halting is essential for:

- Preventing incorrect information from reaching users
- Stopping error propagation in multi-stage pipelines
- Protecting against costly or irreversible actions based on faulty data

Complete Example: Document Analysis with Lazy Verification

```python
## agents/fact_checker.py
from openai import Agent
from ..tools import search_knowledge_base, verify_citation

instructions = """You are a fact verification specialist.

Your task:
1. Identify factual claims in the provided output
2. Verify each claim against available sources using your tools
3. Report any claims that are incorrect or cannot be verified

Be thorough but efficient. Focus on verifiable facts, not
```

```
    opinions
or subjective statements.

Output format:
- List of verified claims (VERIFIED)
- List of unverified or incorrect claims (UNVERIFIED/
    INCORRECT)
- Overall assessment: PASS or FAIL
"""

fact_checker = Agent(
    name="fact_checker",
    model="gpt-5",  # Use capable model for verification
    instructions=instructions,
    tools=[search_knowledge_base, verify_citation],
    response_format=VerificationResult
)

# manager.py
class DocumentAnalysisManager:
    """Manager with lazy verification."""

    def __init__(self):
        self.primary_agent = analysis_agent
        self.verification_agent = fact_checker
        self.verification_frequency = "every_step"  # or "
    periodic", "final"

    def run(self, document: str) -> AnalysisReport:
        """Analyze document with verification."""

        # Extract claims
        claims = self.primary_agent.run(
            f"Extract key claims from: {document}"
        )

        # Verify (series mode for critical application)
        if self.verification_frequency == "every_step":
            verification = self.verification_agent.run(
    claims)
            halt_if_critical(verification)

            if verification.errors_found:
                # Provide feedback for correction
                claims = self._request_correction(claims,
    verification)

        # Continue with verified claims
        analysis = self.analyze_claims(claims)

        return analysis

    def _request_correction(self, output, verification):
        """Request corrected output from primary agent."""
        feedback = f"""Your output had verification errors:
        {verification.error_details}
```

```
64
65          Please provide a corrected version."""
66
67          corrected = self.primary_agent.run(feedback)
68
69          # Re-verify (limit iterations to prevent infinite
    loops)
70          return corrected
```

Providing Feedback to the Originating Agent

When the verification agent finds errors, we have several options for providing feedback:

Option 1: Simple Error Report

Return the error to the calling context:

```
1 if verification.errors_found:
2     return {
3         "status": "error",
4         "message": "Verification failed",
5         "details": verification.error_details
6     }
```

Option 2: Feedback Loop to Original Agent

Provide specific error information to the agent for self-correction:

```
1 def feedback_loop(
2     self,
3     agent,
4     original_input: str,
5     output,
6     verification: VerificationResult,
7     max_iterations: int = 3
8 ) -> str:
9     """Iteratively correct output based on verification
    feedback."""
10
11     for iteration in range(max_iterations):
12         if not verification.errors_found:
13             return output
14
15         # Construct feedback message
16         feedback = f"""Your previous response had the
    following errors:
17
18         {self._format_errors(verification.error_details)}
19
20         Original input: {original_input}
21         Your previous response: {output}
```

```
22
23          Please provide a corrected response that addresses
    these errors."""
24
25          # Agent attempts correction
26          output = agent.run(feedback)
27
28          # Re-verify
29          verification = self.verification_agent.run(output)
30
31      # Max iterations reached
32      if verification.errors_found:
33          raise MaxIterationsException(
34              f"Could not produce correct output after {
    max_iterations} attempts"
35          )
36
37      return output
38
39  def _format_errors(self, errors: list[str]) -> str:
40      """Format errors for clear feedback."""
41      formatted = []
42      for i, error in enumerate(errors, 1):
43          formatted.append(f"{i}. {error}")
44      return "\n".join(formatted)
```

Option 3: Correction by Verification Agent

Have the verification agent not just identify errors but correct them:

```
1  ## agents/correcting_verifier.py
2  correcting_instructions = """You are a fact verification
       and correction specialist.
3
4  Your task:
5  1. Identify factual claims in the provided output
6  2. Verify each claim against available sources
7  3. For incorrect claims: provide the correct information
8  4. Return a corrected version of the output
9
10 Do not change style or structure unnecessarily--only fix
       factual errors."""
11
12 correcting_verifier = Agent(
13     name="correcting_verifier",
14     model="gpt-5",
15     instructions=correcting_instructions,
16     tools=[search_knowledge_base, verify_citation]
17 )
18
19 def run_with_auto_correction(self, user_input: str) -> str:
20     """Run with automatic error correction."""
21
22     # Primary agent generates output
```

```
    output = self.primary_agent.run(user_input)

    # Verifier checks and corrects
    corrected_output = self.correcting_verifier.run(
        f"Verify and correct if needed: {output}"
    )

    return corrected_output
```

Which Feedback Approach to Use?

- **Simple error report**: When human review is required, or errors should halt processing
- **Feedback loop**: When the original agent has the context needed to correct itself
- **Correction by verifier**: When corrections are straightforward and don't require original context

5.2.2 The Overseer: Parallel Spot Checks

While the Lazy Agent Check operates at each step (or periodically), the **Overseer pattern** performs less frequent but more comprehensive checks. The Overseer is a more capable agent that reviews the work of other agents from a higher-level perspective.

Using a Smarter Model and Wider Context for Periodic Review

The Overseer pattern has distinct characteristics:

- **Smarter model**: Uses a more capable (often more expensive) model than the primary agents
- **Wider context**: Reviews multiple steps or the entire pipeline output
- **Random intervals**: Performs spot checks rather than checking every step
- **Parallel operation**: Runs alongside the main pipeline without blocking it
- **Comprehensive analysis**: Looks for patterns, consistency, and high-level errors

When to Use the Overseer
The Overseer pattern is ideal for:

- Long-running pipelines where checking every step is too expensive
- Systems where occasional errors are acceptable but patterns of errors are not
- Quality assurance in production systems
- Detecting subtle errors that require broader context

Implementation

```python
import random
from datetime import datetime

class OverseerAgent:
    """High-level oversight agent for quality assurance."""

    def __init__(self, check_probability: float = 0.1):
        """
        Initialize overseer.

        Args:
            check_probability: Probability of checking any given operation
                               (0.1 = 10% of operations checked)
        """
        self.check_probability = check_probability
        self.overseer = Agent(
            name="overseer",
            model="gpt-5",  # Use most capable model
            instructions=self._get_overseer_instructions(),
            tools=[search_knowledge_base, access_logs, query_database]
        )
        self.findings = []

    def _get_overseer_instructions(self) -> str:
        return """You are a senior quality assurance overseer.

        Your role:
        1. Review agent outputs for accuracy, consistency, and quality
        2. Examine the full context: inputs, intermediate steps, outputs
        3. Identify patterns of errors or quality degradation
        4. Provide actionable feedback for system improvement

        You have access to:
        - Knowledge bases for fact verification
        - System logs for understanding context
        - Database for historical comparison

        Focus on:
        - Factual accuracy
        - Internal consistency
        - Alignment with system goals
        - Potential edge cases or failure modes

        Provide detailed reports when issues are found."""

    def should_check(self) -> bool:
        """Determine if this operation should be checked."""
```

```
    "
48          return random.random() < self.check_probability
49
50      async def spot_check(
51          self,
52          operation_id: str,
53          input_context: str,
54          intermediate_steps: list[dict],
55          final_output: str
56      ) -> Optional[OverseerReport]:
57          """
58          Perform a spot check on a completed operation.
59
60          Args:
61              operation_id: Unique identifier for this
    operation
62              input_context: Original user input
63              intermediate_steps: All intermediate agent
    outputs
64              final_output: Final system output
65
66          Returns:
67              Overseer report if issues found, None otherwise
68          """
69          if not self.should_check():
70              return None  # Skip this check
71
72          # Compile comprehensive context
73          context = self._compile_context(
74              input_context,
75              intermediate_steps,
76              final_output
77          )
78
79          # Overseer analyzes with full context
80          review = await self.overseer.run_async(
81              f"""Review this operation for quality and
    accuracy:
82
83              {context}
84
85              Provide a detailed assessment."""
86          )
87
88          # Parse review results
89          report = self._parse_review(review, operation_id)
90
91          if report.issues_found:
92              self.findings.append(report)
93              await self._handle_issues(report)
94
95          return report
96
97      def _compile_context(
98          self,
99          input_context: str,
```

```
        intermediate_steps: list[dict],
        final_output: str
    ) -> str:
        """Compile full context for overseer review."""
        context_parts = [
            "=== ORIGINAL INPUT ===",
            input_context,
            "",
            "=== INTERMEDIATE STEPS ===",
        ]

        for i, step in enumerate(intermediate_steps, 1):
            context_parts.append(f"\nStep {i}: {step['agent_name']}")
            context_parts.append(f"Output: {step['output']}")

        context_parts.extend([
            "",
            "=== FINAL OUTPUT ===",
            final_output
        ])

        return "\n".join(context_parts)

    async def _handle_issues(self, report: OverseerReport) -> None:
        """Handle issues found by overseer."""
        if report.severity == "critical":
            # Critical issues: halt and alert
            await self._send_alert(
                f"Critical issue found in operation {report.operation_id}"
            )
            # Could trigger rollback or manual review

        elif report.severity == "high":
            # High severity: log and notify
            logger.error(f"High-severity issue: {report.summary}")

        else:
            # Medium/low: log for analysis
            logger.warning(f"Quality issue: {report.summary}")

    def get_quality_metrics(self) -> dict:
        """Aggregate overseer findings for quality monitoring."""
        if not self.findings:
            return {
                "checks_performed": 0,
                "issues_found": 0,
                "error_rate": 0.0
            }

```

```
        total_checks = len(self.findings)
        issues_found = sum(1 for f in self.findings if f.issues_found)

        return {
            "checks_performed": total_checks,
            "issues_found": issues_found,
            "error_rate": issues_found / total_checks,
            "severity_breakdown": self._count_by_severity(),
            "common_issues": self._identify_patterns()
        }

    def _count_by_severity(self) -> dict:
        """Count issues by severity level."""
        severity_counts = {"low": 0, "medium": 0, "high": 0, "critical": 0}
        for finding in self.findings:
            if finding.issues_found:
                severity_counts[finding.severity] += 1
        return severity_counts

    def _identify_patterns(self) -> list[str]:
        """Identify common patterns in overseer findings."""
        # Analyze findings to identify recurring issues
        # (Implementation depends on report structure)
        return []
```

Integration with MUTTA Manager

```
class ProductionManager:
    """Manager with overseer integration."""

    def __init__(self):
        self.primary_pipeline = ResearchPipeline()
        self.overseer = OverseerAgent(check_probability=0.1)

    async def run(self, user_input: str) -> str:
        """Run pipeline with overseer monitoring."""

        # Track all steps for potential overseer review
        operation_id = generate_operation_id()
        intermediate_steps = []

        # Execute primary pipeline
        try:
            # Step 1
            result1 = self.primary_pipeline.step1(user_input)
            intermediate_steps.append({
                "agent_name": "step1",
                "output": result1
```

```
            })

            # Step 2
            result2 = self.primary_pipeline.step2(result1)
            intermediate_steps.append({
                "agent_name": "step2",
                "output": result2
            })

            # ... more steps ...

            final_output = self.primary_pipeline.finalize(
    result2)

        finally:
            # Asynchronously trigger overseer spot check
            # (doesn't block returning results to user)
            asyncio.create_task(
                self.overseer.spot_check(
                    operation_id,
                    user_input,
                    intermediate_steps,
                    final_output
                )
            )

        return final_output
```

Overseer vs. Lazy Agent Check: Comparison

Aspect	Lazy Agent Check	Overseer
Frequency	Every step or periodic	Random spot checks
Model	Same as primary	More capable
Context	Single step	Multiple steps
Timing	Synchronous or parallel	Asynchronous
Purpose	Catch errors early	Quality assurance
Cost	Moderate	Low (infrequent)
Coverage	High	Sampled

Use both patterns together for comprehensive quality assurance: Lazy Agent Checks for critical steps, Overseer for overall quality monitoring.

5.3 Strategy Selection for Critical Actions

Not all actions require the same level of verification. Some actions are reversible and low-risk; others are irreversible and critical. This section explores how to choose verification strategies based on action criticality.

5.3.1 Non-Reversible Critical Actions vs. Reversible Non-Critical Actions

Actions exist on a spectrum from completely reversible to entirely irreversible.

Reversible Actions
Reversible actions can be undone without significant consequences:

- **Generating a report**: Can regenerate if errors are found
- **Creating a draft**: Can edit or delete
- **Local file operations**: Can be undone or corrected
- **Internal state updates**: Can be rolled back

For reversible actions:

- Lightweight verification is acceptable
- Overseer spot-checks may be sufficient
- Can use parallel verification (catch errors later)
- Accept some error rate and fix post-hoc

Non-Reversible Critical Actions
Non-reversible actions have permanent or difficult-to-reverse consequences:

- **Sending emails to customers**: Can't unsend
- **Processing financial transactions**: Refunds are costly and complex
- **Deleting data**: May be unrecoverable
- **Publishing content**: Public exposure is hard to retract
- **Medical decisions**: Patient safety consequences
- **Legal filings**: Official records

For critical actions:

- Mandatory verification before execution
- Series mode (block until verified)
- Multiple verification layers
- Human-in-the-loop for highest criticality
- Zero-tolerance error policy

5.3.2 The Necessity of Singular Checks Before Critical Steps

For critical actions, you should implement a **singular check**—a mandatory verification immediately before execution. This check acts as a final gate, ensuring that no error slips through.

The Singular Check Pattern

```python
def execute_critical_action(
    self,
    action: CriticalAction,
    context: dict
) -> ActionResult:
    """
    Execute critical action with mandatory singular check.

    This is the FINAL verification before irreversible action.
    """

    # The singular check: comprehensive verification
    check_result = self._singular_check(action, context)

    if not check_result.passed:
        # Halt: do not execute
        logger.critical(
            f"Singular check failed for critical action: {action.name}"
        )
        return ActionResult(
            status="aborted",
            reason=check_result.failure_reason
        )

    # All checks passed: execute
    logger.info(f"Executing critical action: {action.name}")
    return self._execute(action)

def _singular_check(
    self,
    action: CriticalAction,
    context: dict
) -> CheckResult:
    """
    Comprehensive final check before critical action.

    This check should verify:
    1. Action parameters are valid
    2. Action is appropriate given context
    3. No obvious errors in reasoning leading to action
    4. Action consequences are acceptable
    """

    # Check 1: Parameter validation
    if not self._validate_parameters(action):
        return CheckResult(
            passed=False,
            failure_reason="Invalid parameters"
        )

    # Check 2: Contextual appropriateness
    appropriateness = self.appropriateness_agent.run(
```

```
        f"""Is this action appropriate?

        Context: {context}
        Proposed action: {action.description}
        Parameters: {action.parameters}

        Respond: YES or NO with explanation."""
    )

    if not appropriateness.is_appropriate:
        return CheckResult(
            passed=False,
            failure_reason=f"Inappropriate: {appropriateness.explanation}"
        )

    # Check 3: Consequence assessment
    consequences = self.assess_consequences(action)

    if consequences.risk_level > action.acceptable_risk:
        return CheckResult(
            passed=False,
            failure_reason=f"Risk too high: {consequences.details}"
        )

    # All checks passed
    return CheckResult(passed=True)
```

Timing of Singular Checks
Singular checks should occur:

- **Immediately before execution**: Not earlier in the pipeline
- **After all processing is complete**: Full context available
- **In series mode**: Block execution until check passes
- **With no timeout**: Take as long as needed for thorough verification

Example: Email Campaign System

```
class EmailCampaignManager:
    """Manage email campaigns with critical action protection."""

    def send_campaign(
        self,
        campaign: Campaign,
        recipient_list: list[str]
    ) -> CampaignResult:
        """Send email campaign with singular check."""

        # Generate email content
```

```
        email_content = self.content_agent.run(campaign.brief)

        # Prepare campaign
        prepared = self._prepare_campaign(email_content, recipient_list)

        # SINGULAR CHECK before sending to thousands of recipients
        check = self._singular_check_campaign(prepared)

        if not check.passed:
            return CampaignResult(
                status="aborted",
                reason=check.failure_reason,
                emails_sent=0
            )

        # Check passed: execute
        return self._execute_send(prepared)

    def _singular_check_campaign(
        self,
        prepared: PreparedCampaign
    ) -> CheckResult:
        """Final check before sending campaign."""

        # Check 1: Content quality
        quality_check = self.quality_agent.run(
            f"Review this email for errors: {prepared.content}"
        )

        if quality_check.errors_found:
            return CheckResult(
                passed=False,
                failure_reason=f"Content errors: {quality_check.errors}"
            )

        # Check 2: Recipient list validation
        if not self._validate_recipients(prepared.recipients):
            return CheckResult(
                passed=False,
                failure_reason="Invalid recipient list"
            )

        # Check 3: Spam/compliance check
        compliance = self.compliance_checker.run(prepared.content)

        if not compliance.compliant:
            return CheckResult(
                passed=False,
                failure_reason=f"Compliance violation: {
```

```
        compliance.issues}"
61              )
62
63          # Check 4: Manual approval for large sends
64          if len(prepared.recipients) > 10000:
65              # Require human approval
66              approval = self._request_human_approval(
        prepared)
67              if not approval.granted:
68                  return CheckResult(
69                      passed=False,
70                      failure_reason="Human approval denied"
71                  )
72
73          return CheckResult(passed=True)
```

5.4 Logical Fault Checking

The verification techniques discussed so far focus on factual accuracy—checking claims against knowledge bases. However, another category of errors requires different treatment: logical errors.

5.4.1 Handling Mathematical Proofs and Purely Logical Tasks

Some agent tasks are primarily logical rather than factual:

- **Mathematical proofs**: Verifying that reasoning steps are valid
- **Code correctness**: Ensuring logic is sound
- **Formal arguments**: Checking that conclusions follow from premises
- **Consistency checking**: Ensuring no contradictions exist

For these tasks, factual grounding isn't enough. We need to verify *logical soundness*.

Example: Mathematical Reasoning

Consider an agent solving a mathematical problem:

```
1 Problem: Prove that for all positive integers n, 1 + 2 +
      ... + n = n(n+1)/2
2
3 Agent's response:
4 "We use mathematical induction.
5
6 Base case: n=1
7 Left side: 1
8 Right side: 1(1+1)/2 = 1
9 Base case holds.
```

```
10
11 Inductive step: Assume 1 + 2 + ... + k = k(k+1)/2
12 Prove for k+1:
13 1 + 2 + ... + k + (k+1) = k(k+1)/2 + (k+1)
14                         = (k(k+1) + 2(k+1))/2
15                         = (k+1)(k+2)/2
16                         = (k+1)((k+1)+1)/2
17
18 Therefore, the formula holds by induction."
```

This proof is logically sound, but how do we verify it programmatically?

5.4.2 Checking Logical Soundness (e.g., Using Lean or Reasoner Agents)

We have several approaches to verifying logical soundness:

Approach 1: Reasoner Agent

Use a specialized reasoning agent to check logical validity:

```python
## agents/logic_checker.py
from openai import Agent

logic_checker_instructions = """You are a formal logic
    verification specialist.

Your task: Analyze reasoning and identify logical errors.

Check for:
1. Invalid inference steps
2. Circular reasoning
3. Contradictions
4. Unsupported assumptions
5. Gaps in reasoning

For each step in the provided reasoning:
- Verify the step follows logically from previous steps
- Identify the inference rule used
- Flag any invalid steps

Provide detailed analysis of any logical errors found."""

logic_checker = Agent(
    name="logic_checker",
    model="gpt-5",  # Use strong reasoning model
    instructions=logic_checker_instructions
)

def verify_logical_reasoning(reasoning: str) ->
    VerificationResult:
    """Verify logical soundness of reasoning."""

    result = logic_checker.run(
```

```
        f"Verify the logical soundness of this reasoning:\n\n{reasoning}"
    )

    return result
```

Approach 2: Formal Verification with Lean

For mathematical proofs, use a formal proof assistant like Lean:

```
@function_tool
def verify_with_lean(proof: str) -> dict:
    """
    Verify a mathematical proof using Lean theorem prover.

    Args:
        proof: Proof in Lean syntax

    Returns:
        Verification result with success status and errors
    """
    import subprocess
    import tempfile

    # Write proof to temporary file
    with tempfile.NamedTemporaryFile(mode='w', suffix='.lean', delete=False) as f:
        f.write(proof)
        proof_file = f.name

    try:
        # Run Lean to verify proof
        result = subprocess.run(
            ['lean', proof_file],
            capture_output=True,
            text=True,
            timeout=30
        )

        if result.returncode == 0:
            return {
                "verified": True,
                "message": "Proof verified successfully"
            }
        else:
            return {
                "verified": False,
                "errors": result.stderr
            }

    except subprocess.TimeoutExpired:
        return {
            "verified": False,
            "errors": "Verification timeout (proof too
```

```
    complex)"
44          }
45
46      finally:
47          os.unlink(proof_file)
48
49  # Agent that generates Lean proofs
50  lean_proof_agent = Agent(
51      name="lean_proof_agent",
52      model="gpt-5",
53      instructions="""You are a mathematician who writes
    formal proofs in Lean.
54
55      When given a theorem to prove:
56      1. Write a formal proof in Lean syntax
57      2. Use the verify_with_lean tool to check your proof
58      3. If verification fails, analyze errors and revise
59      4. Iterate until proof verifies successfully
60
61      Only return proofs that have been verified by Lean.""",
62      tools=[verify_with_lean]
63  )
```

Approach 3: Code Correctness Checking

For logical errors in code reasoning:

```
1   @function_tool
2   def check_code_logic(code: str, test_cases: list[dict]) ->
    dict:
3       """
4       Verify code logic by running test cases.
5
6       Args:
7           code: Python code to verify
8           test_cases: List of {"input": ..., "expected_output
    ": ...}
9
10      Returns:
11          Verification results
12      """
13      import sys
14      from io import StringIO
15
16      passed = []
17      failed = []
18
19      for i, test in enumerate(test_cases):
20          try:
21              # Execute code in isolated namespace
22              namespace = {}
23              exec(code, namespace)
24
25              # Assuming code defines a function named '
```

```
    solution'
            if 'solution' not in namespace:
                return {
                    "verified": False,
                    "error": "Code must define a 'solution'
    function"
                }

            # Run test
            actual_output = namespace['solution'](test['
    input'])

            if actual_output == test['expected_output']:
                passed.append(i)
            else:
                failed.append({
                    "test_case": i,
                    "input": test['input'],
                    "expected": test['expected_output'],
                    "actual": actual_output
                })

        except Exception as e:
            failed.append({
                "test_case": i,
                "error": str(e)
            })

    return {
        "verified": len(failed) == 0,
        "passed": len(passed),
        "failed": len(failed),
        "failures": failed
    }

code_verification_agent = Agent(
    name="code_verifier",
    model="gpt-5",
    instructions="""You verify code correctness.

    When reviewing code:
    1. Analyze the logic for potential errors
    2. Generate comprehensive test cases
    3. Use check_code_logic to verify
    4. Report any failures with explanations

    Look for common logical errors:
    - Off-by-one errors
    - Edge case handling
    - Incorrect conditionals
    - Wrong loop bounds""",
    tools=[check_code_logic]
)
```

Approach 4: Consistency Checking

For general logical consistency:

```python
def check_consistency(statements: list[str]) -> dict:
    """
    Check if a set of statements is logically consistent.

    Uses an LLM to identify contradictions.
    """

    consistency_prompt = f"""Analyze these statements for logical consistency:

{chr(10).join(f'{i+1}. {s}' for i, s in enumerate(statements))}

Identify any contradictions or logical inconsistencies.
If statements are consistent, respond "CONSISTENT".
If inconsistent, explain the contradiction."""

    result = logic_checker.run(consistency_prompt)

    return {
        "consistent": "CONSISTENT" in result.upper(),
        "analysis": result
    }

# Example usage
statements = [
    "All employees must work from the office.",
    "Remote work is available to all employees.",
    "The office is closed."
]

consistency = check_consistency(statements)
# Would identify contradictions
```

Combining Logical and Factual Verification

Complex tasks often require both types of verification:

```python
class ComprehensiveVerifier:
    """Verifier that checks both facts and logic."""

    def __init__(self):
        self.fact_checker = fact_checking_agent
        self.logic_checker = logic_checking_agent

    def verify(self, output: str, context: dict) -> VerificationResult:
        """Comprehensive verification."""

        # Check facts
        fact_result = self.fact_checker.run(
            f"Verify factual claims in: {output}"
        )
```

```
15
16          # Check logic
17          logic_result = self.logic_checker.run(
18              f"Verify logical soundness of: {output}"
19          )
20
21          # Combine results
22          issues = []
23
24          if fact_result.errors_found:
25              issues.extend(fact_result.errors)
26
27          if logic_result.errors_found:
28              issues.extend(logic_result.errors)
29
30          return VerificationResult(
31              passed=len(issues) == 0,
32              fact_check_passed=not fact_result.errors_found,
33              logic_check_passed=not logic_result.
      errors_found,
34              issues=issues
35          )
```

Now that we have explored the main strategies for error detection, correction, and prevention in agent systems, lets review the key points:

- **Grounding principles**: Errors can be factual (contradicting verifiable information) or logical (violating reasoning rules). Grounding means anchoring outputs to verifiable sources of truth.
- **Lazy Agent Check**: A verification agent that checks outputs against knowledge bases using RAG or Navigator patterns. Can operate in series (blocking) or parallel (async) mode with production halting for critical errors.
- **Feedback mechanisms**: Three approaches for handling detected errors: simple error reports, feedback loops to original agents, or correction by the verification agent itself.
- **Overseer pattern**: High-level quality assurance using more capable models with broader context. Performs random spot-checks rather than checking every step, providing statistical quality monitoring.
- **Strategy selection**: Choose verification approach based on action criticality. Reversible actions tolerate lightweight verification; critical/irreversible actions require mandatory singular checks.
- **Logical fault checking**: For mathematical proofs and reasoning tasks, use specialized approaches: reasoner agents, formal verification tools (Lean), code testing, or consistency checking.

Error correction is not a one-size-fits-all solution. The appropriate strategy depends on:

- Error type (factual vs. logical)
- Action criticality (reversible vs. irreversible)
- Performance requirements (latency, cost)
- Quality requirements (acceptable error rate)

A production system typically combines multiple strategies:

- Lazy checks on critical path steps
- Overseer for overall quality monitoring
- Singular checks before irreversible actions
- Logical verification for reasoning-heavy tasks

With the MUTTA architecture (Chapter 2), input alignment (Chapter 3), reusable patterns (Chapter 4), and error correction (this chapter), we have a complete toolkit for building production-grade agent systems. In the final chapter, we'll bring everything together through practical applications and case studies, demonstrating how these principles combine to create robust, reliable agentic systems.

Chapter 6

Practical Applications and Agentic Best Practices

Throughout this book, we have built a comprehensive framework for designing and implementing production-grade AI agent systems. We established the foundations of agent architectures (Chapter 1), introduced the MUTTA pattern for modular service design (Chapter 2), developed techniques for ensuring input quality (Chapter 3), explored reusable patterns that extend agent capabilities (Chapter 4), and implemented robust error correction strategies (Chapter 5).

Now, in this final chapter, we bring everything together through practical application. We will:

1. **Demonstrate complete systems**: Walk through real-world case studies showing how the principles combine in practice
2. **Encode architectural knowledge**: Transform the MUTTA pattern into rules that coding agents can follow
3. **Provide templates and best practices**: Offer reusable starting points for common agent system patterns
4. **Synthesize key principles**: Distill the essential lessons for building robust agentic systems

The goal is not merely to showcase examples, but to provide actionable guidance that you can immediately apply to your own projects. By the end of this chapter, you will have concrete templates, encoded rules for AI coding assistants, and a clear path from concept to production.

6.1 Demonstration and Case Studies

The best way to understand how all the pieces fit together is through complete, working examples. This section presents three case studies of increasing complexity, each demonstrating different aspects of the MUTTA architecture and the patterns we've developed.

6.1.1 Building a Basic Chat Service

We begin with a foundational use case: a conversational agent that can answer questions, maintain context, and access external knowledge. While conceptually simple, this chat service demonstrates core MUTTA principles and serves as a template for more complex systems.

Requirements
Our chat service should:

- Engage in natural conversation with users
- Maintain conversation history for context
- Access a knowledge base to answer domain-specific questions
- Gracefully handle questions outside its knowledge
- Validate inputs and provide helpful error messages

Architecture Overview
Following the MUTTA pattern:

```
chat_service/
        manager.py              # Orchestrates conversation flow
        agents/
            chat_agent.py       # Main conversational agent
            knowledge_agent.py  # RAG-enabled knowledge retrieval agent
        tools.py                # Knowledge base search tools
        utilities.py            # Input validation, history management
        knowledge_base/
            documents/          # Domain-specific documents
```

Step 1: Utilities for Input and History Management

```python
## utilities.py
"""Utility functions for chat service."""

from pydantic import BaseModel, Field
from typing import List, Optional
from datetime import datetime

class Message(BaseModel):
    """A single message in the conversation."""
    role: str = Field(description="'user' or 'assistant'")
    content: str = Field(description="Message content")
    timestamp: datetime = Field(default_factory=datetime.now)

class ConversationHistory:
    """Manages conversation history."""

    def __init__(self, max_messages: int = 20):
        self.messages: List[Message] = []
```

```
        self.max_messages = max_messages

    def add_message(self, role: str, content: str) -> None:
        """Add a message to history."""
        self.messages.append(Message(role=role, content=
    content))

        # Trim if exceeding max
        if len(self.messages) > self.max_messages:
            self.messages = self.messages[-self.
    max_messages:]

    def get_context(self, num_messages: int = 10) -> str:
        """Get recent conversation context as formatted
    string."""
        recent = self.messages[-num_messages:]

        context_parts = []
        for msg in recent:
            context_parts.append(f"{msg.role.upper()}: {msg
    .content}")

        return "\n".join(context_parts)

    def clear(self) -> None:
        """Clear conversation history."""
        self.messages = []

def validate_user_input(user_input: str) -> str:
    """
    Validate and sanitize user input.

    Args:
        user_input: Raw user input

    Returns:
        Cleaned input

    Raises:
        ValueError: If input is invalid
    """
    if not user_input or not user_input.strip():
        raise ValueError("Input cannot be empty")

    # Remove excessive whitespace
    cleaned = " ".join(user_input.split())

    # Check length
    if len(cleaned) > 2000:
        raise ValueError("Input too long (max 2000
    characters)")

    return cleaned
```

Step 2: Tools for Knowledge Access

```python
## tools.py
"""Tools for chat service agents."""

from openai import function_tool
from typing import List, Dict
import os

# Initialize knowledge base (from Chapter 4)
from knowledge_base import SimpleRAGStore

# Load pre-built knowledge base
kb = SimpleRAGStore()
kb.load("knowledge_base/embeddings.pkl")

@function_tool
def search_knowledge_base(query: str, top_k: int = 3) ->
    List[Dict[str, str]]:
    """
    Search the knowledge base for relevant information.

    Args:
        query: Search query
        top_k: Number of results to return

    Returns:
        List of relevant knowledge chunks with sources
    """
    results = kb.retrieve(query, top_k=top_k)

    return [
        {
            "content": r["text"],
            "relevance_score": f"{r['score']:.3f}",
            "source": r["metadata"].get("source", "Unknown"
    )
        }
        for r in results
    ]

@function_tool
def get_current_time() -> str:
    """Get the current date and time."""
    from datetime import datetime
    return datetime.now().strftime("%Y-%m-%d %H:%M:%S")
```

Step 3: Agent Definitions

```python
## agents/chat_agent.py
"""Main conversational agent."""

from openai import Agent
from ..tools import get_current_time

```

```
instructions = """You are a helpful, friendly assistant.

Your role:
- Engage in natural, helpful conversation
- Maintain context from the conversation history
- If asked about specialized topics, suggest using the
    knowledge base
- Be concise but thorough
- Admit when you don't know something

Guidelines:
- Keep responses focused and relevant
- Use the conversation history to maintain context
- Be polite and professional
"""

chat_agent = Agent(
    name="chat_agent",
    model="gpt-5",
    instructions=instructions,
    tools=[get_current_time]
)
```

```
## agents/knowledge_agent.py
"""RAG-enabled knowledge retrieval agent."""

from openai import Agent
from ..tools import search_knowledge_base

instructions = """You are a knowledge specialist with
    access to a curated
knowledge base.

Your role:
- Answer questions using information from the knowledge
    base
- Always cite your sources
- If the knowledge base doesn't contain relevant
    information, say so clearly
- Provide accurate, well-grounded responses

Process:
1. Search the knowledge base for relevant information
2. Synthesize findings from retrieved documents
3. Cite sources for all claims
4. If information is not in the knowledge base, state this
    clearly

Remember: Only make claims you can support with retrieved
    documents."""

knowledge_agent = Agent(
    name="knowledge_agent",
```

```
    model="gpt-5",
    instructions=instructions,
    tools=[search_knowledge_base]
)
```

Step 4: Manager Orchestration

```
## manager.py
"""Chat service manager."""

from pydantic import BaseModel
from typing import Optional
from .agents.chat_agent import chat_agent
from .agents.knowledge_agent import knowledge_agent
from .utilities import (
    validate_user_input,
    ConversationHistory
)

class ChatResponse(BaseModel):
    """Response from chat service."""
    response: str
    used_knowledge_base: bool = False
    sources: list[str] = []

class ChatServiceManager:
    """
    Manages the chat service using MUTTA architecture.

    This manager determines whether to use the general chat
    agent
    or the specialized knowledge agent based on the query.
    """

    def __init__(self):
        """Initialize the chat service."""
        self.history = ConversationHistory(max_messages=20)
        self.general_agent = chat_agent
        self.knowledge_agent = knowledge_agent

    def chat(self, user_input: str) -> ChatResponse:
        """
        Process a chat message.

        Args:
            user_input: User's message

        Returns:
            ChatResponse with agent's reply

        Raises:
            ValueError: If input is invalid
        """
        # Step 1: Validate input
```

```
47          cleaned_input = validate_user_input(user_input)
48
49          # Step 2: Add to history
50          self.history.add_message("user", cleaned_input)
51
52          # Step 3: Determine if specialized knowledge is
    needed
53          needs_knowledge = self._needs_knowledge_base(
    cleaned_input)
54
55          # Step 4: Route to appropriate agent
56          if needs_knowledge:
57              response = self._query_knowledge_agent(
    cleaned_input)
58          else:
59              response = self._query_general_agent(
    cleaned_input)
60
61          # Step 5: Add response to history
62          self.history.add_message("assistant", response.
    response)
63
64          return response
65
66      def _needs_knowledge_base(self, query: str) -> bool:
67          """
68          Heuristic to determine if query requires knowledge
    base.
69
70          In production, could use a classifier agent or
    keywords.
71          """
72          # Simple keyword-based heuristic
73          knowledge_keywords = [
74              "what is", "define", "explain", "how does",
75              "documentation", "specification", "details
    about"
76          ]
77
78          query_lower = query.lower()
79          return any(keyword in query_lower for keyword in
    knowledge_keywords)
80
81      def _query_general_agent(self, user_input: str) ->
    ChatResponse:
82          """Query the general conversational agent."""
83
84          # Get conversation context
85          context = self.history.get_context(num_messages=10)
86
87          # Create prompt with context
88          prompt = f"""Conversation history:
89  {context}
90
91  USER: {user_input}
92
```

```
Respond to the user's message."""

        response = self.general_agent.run(prompt)

        return ChatResponse(
            response=response,
            used_knowledge_base=False
        )

    def _query_knowledge_agent(self, user_input: str) ->
    ChatResponse:
        """Query the knowledge-specialized agent."""

        # Knowledge agent uses RAG, so less history needed
        context = self.history.get_context(num_messages=3)

        prompt = f"""Recent context:
{context}

USER: {user_input}

Answer using the knowledge base. Search for relevant
   information and
provide a grounded response."""

        response = self.knowledge_agent.run(prompt)

        # Extract sources (would need structured output in
    production)
        return ChatResponse(
            response=response,
            used_knowledge_base=True,
            sources=["knowledge_base"]  # Simplified
        )

    def reset(self) -> None:
        """Reset conversation history."""
        self.history.clear()
```

Step 5: Service Interface

```
## __init__.py
"""Chat service public interface."""

from .manager import ChatServiceManager, ChatResponse

def create_chat_service() -> ChatServiceManager:
    """Factory function to create a chat service instance."
    ""
    return ChatServiceManager()

__all__ = ["ChatServiceManager", "ChatResponse", "
    create_chat_service"]
```

Usage Example

```python
## Example usage
from chat_service import create_chat_service

# Initialize service
chat = create_chat_service()

# Conversation
response1 = chat.chat("Hello! How are you?")
print(response1.response)
# "Hello! I'm doing well, thank you for asking. How can I
    help you today?"

response2 = chat.chat("What is the MUTTA architecture?")
print(response2.response)
# Searches knowledge base and provides grounded answer with
     sources

response3 = chat.chat("Can you elaborate on the manager
    component?")
print(response3.response)
# Maintains context from previous question, provides
    detailed answer

# Reset conversation
chat.reset()
```

Key Patterns Demonstrated

This basic chat service demonstrates:

- **MUTTA structure**: Clear separation of manager, agents, tools, utilities
- **Input validation**: Using utilities to ensure clean inputs
- **RAG pattern**: Knowledge agent uses retrieval for grounded responses
- **Routing logic**: Manager decides which agent to use
- **Context management**: Conversation history maintained and provided to agents
- **Clean interfaces**: Public API hides implementation details

6.1.2 Designing Complex Systems (e.g., Automatic Academic Researcher, Marketing Engine)

Building on the chat service foundation, we now explore two complex systems that demonstrate advanced MUTTA patterns, multi-agent coordination, and error correction.

Case Study 1: Automatic Academic Researcher

An automatic academic researcher assists with literature review by finding relevant papers, extracting key information, and synthesizing findings.

System Requirements

- Search academic databases for relevant papers
- Extract key information: methodology, findings, citations
- Identify themes and connections across papers
- Generate structured literature review reports
- Verify claims and citations

Architecture

```
academic_researcher/
          manager.py
          agents/
                search_agent.py           # Finds relevant papers
                extraction_agent.py       # Extracts information from papers
                synthesis_agent.py        # Synthesizes findings
                verification_agent.py     # Verifies citations and claims
                report_agent.py           # Generates final report
          tools.py                       # API tools, PDF processing
          utilities.py                   # Data transformation, validation
          __init__.py
```

Key Implementation Details

```
## manager.py (excerpt)
class AcademicResearchManager:
    """
    Orchestrates academic research pipeline.

    Implements systematic/in-series multi-agent pattern for reliability.
    """

    def run(self, research_query: str) -> ResearchReport:
        """
        Execute complete research pipeline.

        Args:
```

```
            research_query: Research question or topic

        Returns:
            Comprehensive research report
        """
        # Step 1: Validate and refine query
        refined_query = self._refine_query(research_query)

        # Step 2: Search for papers (with error handling)
        try:
            papers = self.search_agent.run(
                f"Find 20 most relevant papers on: {refined_query}"
            )
        except Exception as e:
            raise ResearchException(f"Paper search failed: {e}")

        # Step 3: Extract information from each paper
        extractions = []
        for paper in papers.results:
            extraction = self.extraction_agent.run(
                f"Extract key information from: {paper.title}\n{paper.abstract}"
            )

            # Verify citations (error correction)
            verification = self.verification_agent.run(extraction)
            if verification.errors_found:
                # Provide feedback for correction
                extraction = self._correct_extraction(extraction, verification)

            extractions.append(extraction)

        # Step 4: Synthesize findings across papers
        synthesis = self.synthesis_agent.run(
            f"Synthesize findings from {len(extractions)} papers on {refined_query}"
        )

        # Step 5: Generate final report
        report = self.report_agent.run(
            f"Create literature review report: {synthesis}"
        )

        # Step 6: Final verification (overseer pattern)
        if self.overseer.should_check():
            asyncio.create_task(
                self.overseer.spot_check(
                    research_query,
                    extractions,
                    report
                )
            )
```

```
64
65          return report
```

Patterns Demonstrated

- **Systematic/in-series architecture**: Reliable, structured workflow
- **Navigator pattern**: PDF exploration for information extraction
- **Error correction**: Verification agent checks citations
- **Overseer pattern**: Quality assurance on final output
- **Service depth**: 15 steps (within Rule of 20)

Case Study 2: Marketing Content Engine

A marketing engine generates multi-channel campaigns: blog posts, social media, email sequences—all maintaining brand consistency.

System Requirements

- Generate content for multiple channels (blog, social, email)
- Maintain consistent brand voice across channels
- Optimize content for each channel's constraints and audience
- Generate variations for A/B testing
- Verify compliance and quality before publishing

Architecture

This system uses a *parallel multi-agent* architecture for speed:

```
1  marketing_engine/
2            manager.py
3            agents/
4                  strategy_agent.py         # Develops
       campaign strategy
5                  blog_writer.py            # Writes blog
       content
6                  social_specialist.py      # Creates social
       media posts
7                  email_specialist.py       # Writes email
       sequences
8                  brand_guardian.py         # Ensures brand
       consistency
9                  compliance_checker.py     # Verifies
       compliance
10           tools.py
11           utilities.py
12           __init__.py
```

Key Implementation: Parallel Execution with Brand Consistency

```python
## manager.py (excerpt)
class MarketingEngineManager:
    """
    Marketing content generation with parallel agents.

    Implements parallel multi-agent pattern for speed.
    """

    async def run(self, campaign_brief: str) -> MarketingCampaign:
        """
        Generate complete marketing campaign.

        Args:
            campaign_brief: Campaign objectives and requirements

        Returns:
            Multi-channel marketing campaign
        """
        # Step 1: Develop strategy (sequential)
        strategy = self.strategy_agent.run(campaign_brief)

        # Step 2: Generate content in parallel
        blog_task = asyncio.create_task(
            self.blog_writer.run_async(
                f"Write blog post: {strategy.blog_topic}"
            )
        )

        social_task = asyncio.create_task(
            self.social_specialist.run_async(
                f"Create social posts: {strategy.social_topics}"
            )
        )

        email_task = asyncio.create_task(
            self.email_specialist.run_async(
                f"Write email sequence: {strategy.email_flow}"
            )
        )

        # Wait for all content generation
        blog_content, social_content, email_content = await asyncio.gather(
            blog_task, social_task, email_task
        )

        # Step 3: Brand consistency check (critical)
        brand_check = self.brand_guardian.run(
            f"""Review for brand consistency:
```

```
            Blog: {blog_content}
            Social: {social_content}
            Email: {email_content}
            """
        )

        if not brand_check.consistent:
            # Halt and request revisions
            raise BrandInconsistencyException(brand_check.
    issues)

        # Step 4: Compliance verification (singular check
    before publishing)
        compliance = self.compliance_checker.run({
            "blog": blog_content,
            "social": social_content,
            "email": email_content
        })

        if not compliance.approved:
            # Critical: cannot publish
            raise ComplianceException(compliance.violations
    )

        # All checks passed: return campaign
        return MarketingCampaign(
            strategy=strategy,
            blog=blog_content,
            social=social_content,
            email=email_content,
            verified=True
        )
```

Patterns Demonstrated

- **Parallel multi-agent**: Blog, social, email generated simultaneously
- **Handoff pattern**: Strategy agent hands off to content specialists
- **Singular check**: Mandatory compliance check before publishing
- **Critical action protection**: Publishing is irreversible, requires verification
- **Brand consistency**: Cross-cutting concern verified centrally

Comparison of Architectures

System	Pattern	Priority	Error Strategy
Chat Service	Handoff	Generality	Lazy checks
Academic Researcher	Systematic	Reliability	Verification + Overseer
Marketing Engine	Parallel	Speed	Singular checks

Each system uses the MUTTA architecture but adapts the multi-agent pattern and error correction strategy to its specific requirements.

6.2 Agentic Rules and Templates

One of the most powerful applications of the MUTTA architecture is that it can be encoded into rules for AI coding assistants. By teaching coding agents like Cursor, GitHub Copilot, or Windsurf about MUTTA, we enable them to help build agent systems correctly.

6.2.1 Encoding MUTTA Architecture into Coding Agent Rules (Cursor/GitHub Copilot)

The systematic structure of MUTTA makes it ideal for encoding as explicit rules. When a coding agent understands the pattern, it can:

- Generate properly structured MUTTA services
- Suggest appropriate file organization
- Warn when architectural principles are violated
- Provide templates for common patterns

The Challenge Without Encoded Rules
Building agent systems without systematic guidance leads to:

- Agents defined dynamically in manager code (anti-pattern)
- Orchestration logic mixed with agent prompts
- Tools and utilities conflated
- Inconsistent file organization
- Difficulty maintaining and debugging systems

The Solution: Encoding MUTTA as Coding Agent Rules
By providing explicit rules, coding agents can enforce best practices automatically. Here's how to encode MUTTA for Cursor (similar approaches work for other coding assistants):
Step 1: Create .cursor/rules/openai-agents-service.mdc
This is the actual production Cursor rule used for building services with the OpenAI Agents SDK:

```
---
description: Whenever a new agentic service is discussed or
    needed,
             use this rule to guide the creation of the
   service.
alwaysApply: false
---

# OpenAI Agents SDK Service Architecture Guide
```

````
8
9  This rule provides comprehensive guidelines for building
       services using
10 the OpenAI Agents SDK. Follow these patterns to create
       easily extendable,
11 achievable, and modular agent services.
12
13 ## Service Definition
14
15 A **service** is a system that:
16 - Takes an input
17 - Runs through a manager that coordinates multiple agents
18 - Works through a series of steps with that input
19 - Returns a determined output
20
21 Example: A calendar enricher service that takes an email as
       input, checks
22 calendars and weather through agents, and returns an
       enriched email with
23 scheduled actions.
24
25 ## Directory Structure
26
27 ### Base Directory
28 Services should be stored in the `agents_sdk/` directory (
       or user-specified
29 location). Each service follows this structure:
30
31 ```
32 agents_sdk/
33           {service_name}_agents/
34                 __init__.py          # Empty file (required)
35                 manager.py           # Main manager class
36                 agents/              # Directory containing
       all agent files
37                       __init__.py
38                       agent1.py       # Individual agent
       files
39                       agent2.py
40                 tools.py            # Optional: Agent tools
       and functions
41                 utilities.py        # Optional: Helper
       functions for manager
42 ```
43
44 ## File Structure Guidelines
45
46 ### 1. manager.py
47
48 **Must contain:**
49 - A manager class that serves as the main entry point
50 - Imports for agents from the agents/ directory
51 - A main method that takes input and returns output
52 - A wrapper function/class for sync/async compatibility
53
54 **Key Requirements:**
````

```
- Always `await` agent Runners (critical for OpenAI SDK)
- Follow OpenAI SDK documentation for Runners
- Structure operations in logical steps
- Use as few agents as necessary while maintaining quality
- Import agents from agents/ directory (don't initialize in __init__)

**Example Pattern:**
```python
# Import agents from agents directory
from .agents.agent1 import agent1
from .agents.agent2 import agent2

class ServiceManager:
    async def process(self, input_data):
        # Step 1: Initial processing
        # Step 2: Agent coordination using imported agents
        # Step 3: Final output generation

    def sync_wrapper(self, input_data):
        # Sync wrapper for Django/other contexts
```

2. agents/ Directory

Each agent file contains:
- Instructions as a constant string (uppercase recommended)
- Agent definition with model specification
- Pydantic models for structured outputs
- Optional tools list

Model Selection:
- **GPT-4.1**: Basic operations
- **O1 Mini**: Complex reasoning, multiple simultaneous operations
- **O3**: Most complex operations only when absolutely necessary

Agent File Pattern:
```python
AGENT_INSTRUCTIONS = \"\"\"
Detailed system message and developer instructions for the agent.
Clear description of what the agent does and expected behavior.
\"\"\"

# Pydantic models for structured outputs
from pydantic import BaseModel, Field

class AgentResponse(BaseModel):
    field1: str = Field(description="Clear field description")
    field2: Optional[int] = Field(description="Another field")
    # Never use Dict as field type - use nested Pydantic
```
```

```
    models instead

# Agent definition
agent_name = Agent(
    model="gpt-4.1",
    instructions=AGENT_INSTRUCTIONS,
    tools=[tool1, tool2] # if needed
)
```

Critical Rules:
- One agent per file
- No classes except Pydantic models
- Always define structured output models
- Never use `Dict` fields in Pydantic models - use nested Pydantic models
- Include good field descriptions (not extensive, but clear)

3. tools.py (Optional)

Purpose: Define functions that agents can use as tools

Requirements:
- Use `@function_tool` decorator
- Well-documented docstrings (agents read these)
- Pydantic objects as inputs (never dictionaries)
- No nested tool functions with decorators

Critical Nesting Rule:
    Never do this:
```python
@function_tool
def check_single_city(city: str):
    # Single city check

@function_tool
def check_multiple_cities(cities: List[str]):
    for city in cities:
        check_single_city(city)  # This will fail!
```

    Solutions:
1. **Preferred:** Only give batch tool to agent, no decorator on single
2. **Alternative:** Create internal copy without decorator:
```python
def _check_single_city_internal(city: str):
    # Implementation without decorator

@function_tool
def check_single_city(city: str):
    return _check_single_city_internal(city)

@function_tool
def check_multiple_cities(cities: List[str]):
```

```
    for city in cities:
        _check_single_city_internal(city)  # Use internal version
```
```

**Function Requirements:**
- Only include functions intended as agent tools
- No other utility functions (use utilities.py instead)

### 4. utilities.py (Optional)

**Purpose:** Helper functions for manager.py operations

**Contains:**
- Deterministic functions needed by manager
- Data retrieval functions
- Pre/post-processing utilities
- Functions that are NOT given to agents

**When to use:** Rarely necessary - most operations handled by
manager + agents + tools

## Best Practices

### Manager Design
- Think in modular steps
- Use specialized agents for specific tasks
- Chain agents: Agent 1 → Process → Agent 2 → Final Output
- Always structure outputs using Pydantic models
- Always `await` agent Runners

### Agent Instructions
- Provide clear, detailed instructions
- Always specify expected output structure
- Include context about the agent's role in the larger service
- Quality instructions > quantity of agents

### Structured Outputs
- Always use Pydantic BaseModel for agent responses
- Define models in the same agent file
- Use nested Pydantic models instead of Dict fields
- Include clear field descriptions
- Well-fitted models for specific use cases

### Error Handling
- Implement proper error handling in manager
- Use logging for debugging
- Provide meaningful error messages
- Consider fallback strategies

## Integration Guidelines

### Django Integration
```

```
- Create sync wrappers for async agent operations
- Handle Django ORM operations properly

### Testing
- Test manager operations end-to-end
- Test error scenarios

## Service Depth

- Keep service depth     20 sequential steps
- Depth = agent invocations (each containing   10   LLM
   turns)
- Utilities don't count toward depth
- Add error correction if exceeding depth
```

This production-tested rule encapsulates the MUTTA architecture in a format that coding agents can understand and apply. By encoding these patterns explicitly, we enable AI assistants to help build agent systems correctly from the start.

6.2.2 Rules for Maintaining Pristine Logic and Agent Definitions

Beyond the MUTTA structure, certain principles ensure that agent systems remain maintainable and debuggable.

Principle 1: Pristine Manager Logic

The manager should read like a specification:

```
## Good: Clear, step-by-step logic
def run(self, input_data):
    # Validate input
    validated = validate_input(input_data)

    # Extract information
    extracted = self.extraction_agent.run(validated)

    # Verify extraction
    if not self.verification_agent.verify(extracted):
        raise VerificationError("Extraction failed
   verification")

    # Generate report
    report = self.report_agent.run(extracted)

    return report

# Bad: Logic obscured by complexity
def run(self, input_data):
    result = self._process(
        self._transform(
            self._validate_and_extract(input_data)
        )
```

```
    )
    return result if self._check(result) else self._retry(
    input_data)
```
```

**Principle 2: Pristine Agent Definitions**
Agent files should be declarative and self-contained:

```
## Good: Complete, clear agent definition
# agents/email_writer.py
\"\"\"
Email writing specialist agent.

Generates professional email content for customer
    communications.
\"\"\"

from openai import Agent
from ..tools import check_grammar

instructions = \"\"\"You are a professional email writer.

Guidelines:
- Use professional, friendly tone
- Keep emails concise (< 200 words)
- Include clear subject lines
- End with appropriate call-to-action

Always use check_grammar tool before finalizing emails
    .\"\"\"

email_writer = Agent(
    name="email_writer",
    model="gpt-5",
    instructions=instructions,
    tools=[check_grammar]
)

# Bad: Incomplete definition
from openai import Agent

agent = Agent(name="agent", model="gpt-5", instructions="
    Write emails")
```

Principle 3: Explicit Over Implicit
Make architectural decisions explicit:

```
## Good: Explicit routing logic
def route_request(self, request):
    if request.type == "technical":
```
```

```
        return self.technical_agent.run(request)
    elif request.type == "billing":
        return self.billing_agent.run(request)
    else:
        return self.general_agent.run(request)

# Bad: Implicit routing buried in complex logic
def route_request(self, request):
    agent = self._get_agent(request)  # Hidden complexity
    return agent.run(request)
```

Principle 4: Type Safety with Pydantic
Use structured inputs and outputs:

```
from pydantic import BaseModel, Field

class ServiceInput(BaseModel):
    query: str = Field(description="User query")
    max_results: int = Field(default=10, ge=1, le=100)

class ServiceOutput(BaseModel):
    results: list[str]
    metadata: dict

class ServiceManager:
    def run(self, input: ServiceInput) -> ServiceOutput:
        # Type-safe throughout
        ...
```

Benefits of Encoded Rules
With these rules in place, coding agents can:

1. **Generate correct structure**: Create MUTTA services automatically
2. **Prevent mistakes**: Warn when patterns are violated
3. **Suggest improvements**: Recommend refactoring opportunities
4. **Maintain consistency**: Ensure all services follow the same pattern

This dramatically reduces the cognitive load of building agent systems and ensures architectural consistency across projects.

6.3 Conclusion

We have reached the end of our journey through the foundations, patterns, and practices of building robust AI agent systems. Let us synthesize the essential lessons and look toward the future of agentic architectures.

The Core Insight: Architecture Matters

The central thesis of this book is simple but profound: *architecture matters in agent systems as much as it does in traditional software.* While much attention in the AI community focuses on models, prompts, and capabilities, we have shown that systematic architectural patterns are essential for building reliable, maintainable, production-grade agent systems.

The MUTTA architecture embodies this principle. By separating concerns—orchestration (Manager), deterministic logic (Utilities), agent capabilities (Tools), and autonomous behavior (Agents)—we create systems that are:

- **Understandable**: Anyone can read the manager and understand the system's workflow
- **Testable**: Each component can be verified independently
- **Maintainable**: Changes are localized to specific components
- **Reusable**: Agents and tools can be shared across services
- **Evolvable**: The system can grow and adapt without becoming unmaintainable

The Foundation: Principles Before Patterns

Throughout this book, we've emphasized principles over prescriptions:

Service-Oriented Thinking (Chapter 2): Model agent systems as services with clear inputs, outputs, and contracts. Constrain service depth to manageable limits (the Rule of 20).

Input Quality First (Chapter 3): Recognize that errors accumulate across sequential steps. Invest in input alignment using embeddings to minimize initial error.

Extend with Patterns (Chapter 4): Use RAG for knowledge, Navigator for exploration, Code Interpreter for computation, and Tool Selector for scale.

Correct Systematically (Chapter 5): Apply verification strategies appropriate to action criticality—from lightweight oversight to mandatory singular checks.

These principles transcend any particular framework or technology. Whether you use OpenAI's Agents SDK, build framework-free systems, or work with future tools not yet imagined, these principles will guide you toward robust architectures.

The Patterns: Universal and Reusable

The four universal patterns we explored—RAG, Navigator, Code Interpreter, and Tool Selector—address fundamental limitations of language models. They are "universal" because every agent system faces knowledge boundaries, exploration needs, computational requirements, and scaling challenges. They are "reusable" because, once implemented, they can be applied across diverse applications with minimal modification.

Critically, these patterns *compose*. A single agent might use:

- RAG to access domain knowledge
- Navigator to explore a codebase
- Code Interpreter to verify calculations
- Tool Selector to choose from thousands of available actions

The MUTTA architecture provides a framework for composing these patterns cleanly.

The Practice: From Concept to Production

We've moved beyond theory to practice, providing:

- Complete working examples (chat service, academic researcher, marketing engine)
- Encoded rules for AI coding assistants
- Templates for common patterns
- Best practices distilled from production systems

These practical artifacts are intended to accelerate your development. You need not build every system from scratch; start with the templates, adapt the patterns, and apply the principles.

The Future: Evolving with the Field

The field of AI agent systems is evolving rapidly. Models improve, new capabilities emerge, and best practices continue to develop. However, the architectural principles we've explored are more stable than any particular technology:

- Separation of concerns will remain valuable regardless of model capabilities
- Input quality will always affect output quality
- Error correction will remain essential as systems grow more complex
- Systematic architecture will continue to enable maintainability

As models become more capable, the *importance* of architecture may actually increase. More powerful models enable more complex systems, which require more rigorous architecture to remain manageable.

Key Takeaways

If you remember nothing else from this book, remember these principles:

1. **Structure your systems**: Use MUTTA or a similar pattern to enforce separation of concerns
2. **Constrain complexity**: Respect depth limits; add error correction when needed
3. **Align inputs**: Use the embedding heuristic to ensure agents receive appropriate data
4. **Ground outputs**: Verify factual claims and logical soundness appropriate to criticality
5. **Compose patterns**: Combine RAG, Navigator, Code Interpreter, and Tool Selector as needed
6. **Encode knowledge**: Teach your coding agents to build systems correctly

7. **Test systematically**: Verify components independently and integration thoroughly

8. **Iterate deliberately**: Monitor, measure, and refine based on actual behavior

A Call to Action

Building robust AI agent systems is both an art and a science. The science lies in understanding the mathematical properties of error propagation, the mechanics of embeddings, and the logic of verification. The art lies in selecting appropriate patterns, balancing trade-offs, and designing systems that are both powerful and maintainable.

You now have the tools, patterns, and principles to build production-grade agent systems. The next step is yours: apply these lessons to your own projects. Start with a simple service following the MUTTA pattern. Add RAG when you need knowledge. Implement verification appropriate to your needs. Encode your learnings as rules for your coding agents.

As you build, remember that every great system begins with a clear architecture and evolves through disciplined iteration. The agent systems of tomorrow will be built by developers who understand not just how to prompt models, but how to architect systems.

Build deliberately. Build systematically. Build robustly.

Final Thoughts

The transition from single-shot LLM calls to sophisticated multi-agent systems represents a paradigm shift in how we build AI applications. Just as the shift from procedural to object-oriented programming required new architectural patterns, the shift to agentic systems requires new patterns like MUTTA.

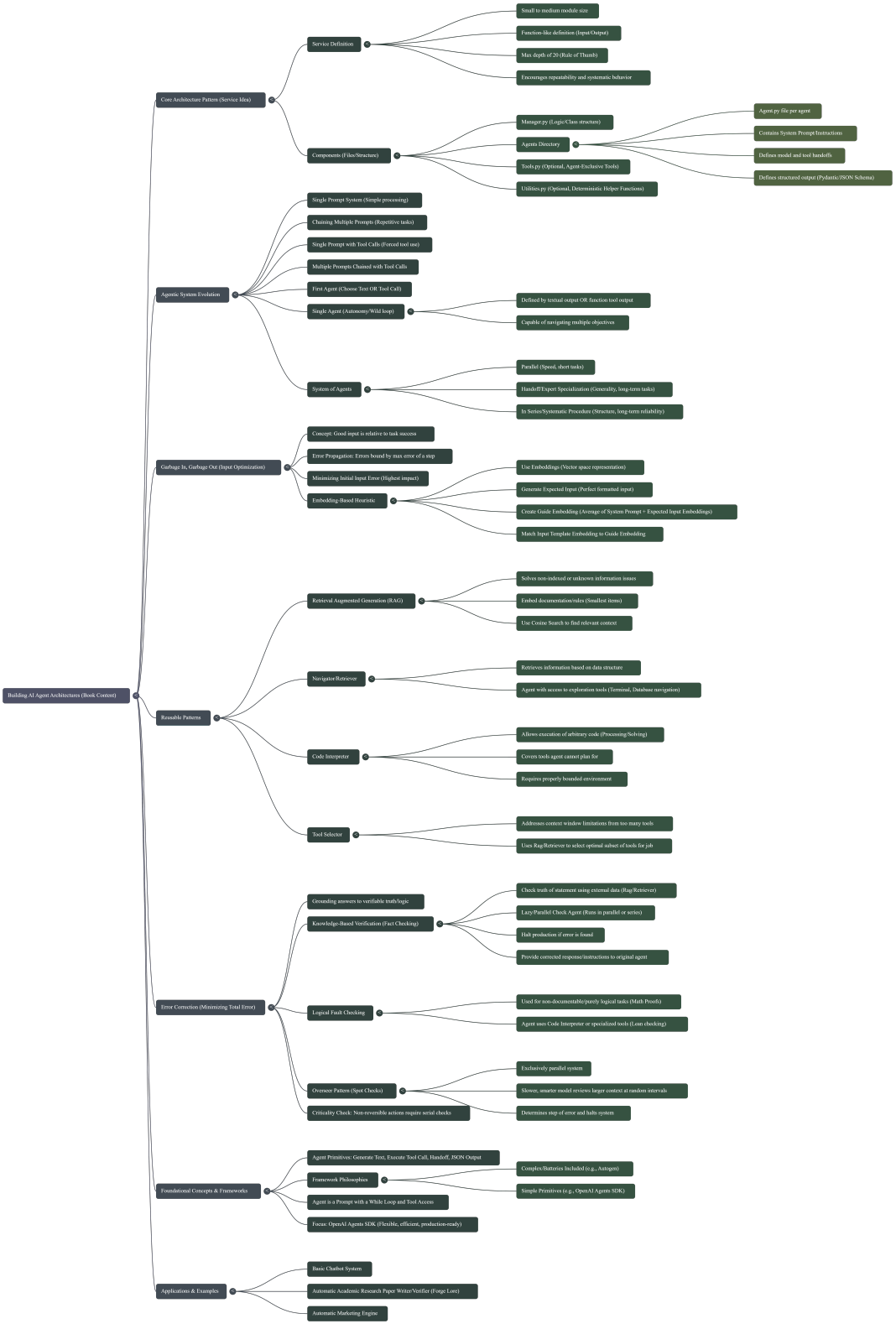
We are still in the early days of this paradigm. The patterns and practices in this book represent current best practices, but the field will continue to evolve. What will not change is the fundamental truth that architecture—clear, systematic, maintainable architecture—is essential for building systems that work reliably in production.

You are now equipped to be part of shaping this evolution. The principles you've learned will serve you regardless of how the technology changes. The patterns you've mastered will adapt to new capabilities. The discipline you've developed will guide you through future challenges.

Go forth and build remarkable agent systems. The future of AI is agentic, and you are ready to build it.

The End

Book Concept Mind Map



Visual representation of the book's core concepts and their relationships

About this book

Agentic AI Systems: Foundations, Patterns, and Architectures provides a comprehensive guide to building production-grade AI agent systems using modern architectural patterns. The book introduces the MUTTA (Manager, Utilities, Tools, and Agents) architecture—a systematic approach to structuring agent services that emphasizes modularity, maintainability, and reliability. Through six comprehensive chapters, readers learn foundational concepts, practical patterns, error correction strategies, and best practices.

Key topics include:

- Multi-agent system architectures (parallel, handoff, systematic)
- Service-oriented design for AI agents
- Input quality and error propagation
- Universal patterns: RAG, Navigator, Code Interpreter, Tool Selector
- Comprehensive error correction strategies • Real-world case studies and implementations
- Encoding architectural knowledge for AI coding assistants

Whether you're building conversational AI, autonomous research assistants, or complex multi-agent workflows, this book provides the architectural foundation for creating robust, maintainable systems

Build deliberately. Build systematically. Build robustly