A PRACTICAL GUIDE TO AI AGENTS

Building AI Agents from Scratch

A Hands-On Guide with Python

Yanfei Li

A PRACTICAL GUIDE WITH PYTHON

Building AI Agents from Scratch

A Hands-On Guide with Python

Yanfei Li

First edition · 2026

Building AI Agents from Scratch

A Hands-On Guide with Python

First edition, 2026.

Examples are educational. Some are study fragments; others are complete programs with documented dependencies. Live integrations require your own accounts and may incur charges. Model identifiers and APIs can change.

Code and technical references were reviewed in September 2026. Local checks do not verify every external integration or production deployment.

Companion code

The companion package follows the chapter folders named throughout this book. Refer to the download instructions provided with your purchased edition. Start with the package README and Chapter 3. Code filenames shown in the chapters refer to that package.

1 What Are AI Agents?

Companion examples: chapter-01. See the setup and example status.

1.1 Defining Agents vs. Chatbots vs. Automation Scripts

An **agent** is a system that selects actions, observes their effects, and uses those observations to pursue a goal. Agents existed before large language models: a game-playing reinforcement learning agent and a robot controller are also agents. This book concentrates on **LLM agents**, whose model selects some of the next steps while application code controls what can actually run.

The useful distinction is who controls the next step. A workflow fixes the sequence and branches in code. An agent lets a model choose among permitted actions based on the task and observations. Real applications often combine both: a fixed authentication and approval workflow surrounds a model-driven research loop.

Design	Who selects the next operation?	Example
Conventional automation	Application code, including retries and conditional branches	Nightly database backup
LLM workflow	Application code schedules model and tool calls	Classify a ticket, then draft a response
LLM agent	Model chooses some actions; runtime validates and executes them	Investigate a ticket using several sources
Chat or voice interface	An interface choice; any of the above can sit behind it	Text chat or a spoken assistant

A chatbot can use tools and persistent memory. A script can recover from failures. A voice interface does not determine whether a system plans. These categories overlap; product names alone do not establish agency.

Consider a weather request. A model without a weather source should say it cannot check current conditions. A fixed workflow can call a weather API for every request. An agent can decide that the request needs current observations, call the weather tool, and answer using the location, units, and observation time returned. The example result is a fixture, not a live weather report:

```
{"location": "Seattle, WA", "temperature_f": 52, "conditions": "Cloudy",
"observed_at": "2026-01-15T12:00:00Z"}
```

For a trip-planning request, an agent might check forecasts, compare activities, and prepare an itinerary. A proposed booking is a separate action: the application checks the user's authorization for the concrete dates, destination, recipients, and price before execution. It reports a reservation only after receiving a booking confirmation. Generating a sentence about an action does not perform it.

1.1.1 Why Use an Agent—and When to Use a Workflow?

Use an agent when the next useful step depends on information that cannot be enumerated cheaply in advance: investigating an unfamiliar repository, reconciling conflicting documents, or choosing follow-up queries after a failed lookup. Tools supply current data and exact computations; the model helps interpret a varied request and choose how to proceed.

Start with a simpler baseline. A known calculation needs a function. A fixed process needs a workflow. A document question may need one retrieval-and-answer pass. Add a loop only if it improves measured task completion enough to justify extra cost, latency, and operational complexity. More autonomy is not a quality metric.

Before building, write down the input, allowed tools, evidence required for success, budget, and fallback. For our first agent: accept an arithmetic question, use a bounded calculator, return the observed result, stop within eight model calls, and report an explicit failure status when the budget is exhausted.

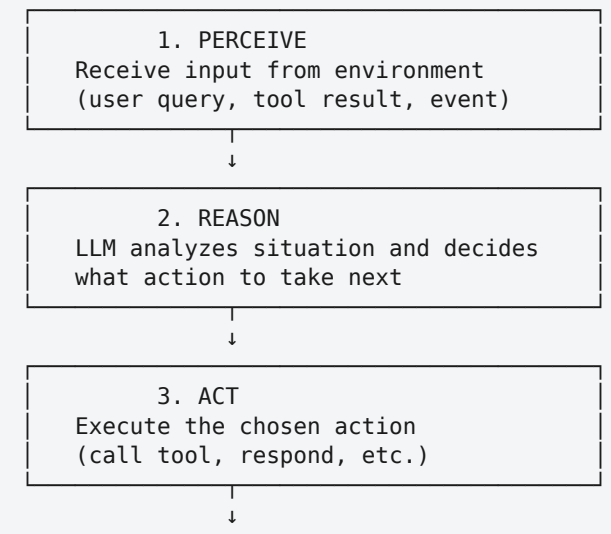
1.1.2 Your Learning Path

Chapters 2–4 build the first working loop. Chapters 5–9 add dependable tools, protocols, memory, planning, and retrieval. Chapters 10–13 compare multi-agent patterns against a single-agent baseline. Chapters 14–16 measure and operate the system. Chapters 17–21 add specialized interfaces and capabilities; Chapter 22 shows how to maintain it as the ecosystem changes.

Keep a small evaluation set from the start. At each stage, demonstrate a successful task, a malformed input, a tool failure, and a budget stop. By the end, you should be able to explain both what your agent accomplished and what evidence supports that conclusion.

1.2 The Agent Loop: Perceive → Reason → Act → Observe

For the LLM agents in this book, the following cycle is a useful implementation model:



```
┌─────────────────────────────────────────┐
│         4. OBSERVE                      │
│   Process the result, decide if done    │
│   or continue to next iteration         │
└─────────────────────────────────────────┘
                  │
                  ↓
                  └──→ Loop back to PERCEIVE
```

Let's break down each phase:

1.2.1 1. Perceive

The agent receives input from its environment. This could be: - **Initial query:** "Find all Python files that import pandas and summarize their purpose" - **Tool result:** `{"files_found": ["analysis.py", "data_prep.py"], "count": 2}` - **Sensor data:** Temperature reading, stock price, incoming email - **Event notification:** Webhook received, file uploaded, deadline approaching

The perception phase is purely input—the agent gathers information about the current state of the world.

1.2.2 2. Reason

This is where the LLM shines. Given the current situation (the original goal + conversation history + latest observation), the agent decides what to do next. The reasoning process might look like:

> **Illustrative decision explanation, not a transcript of hidden model reasoning:**
>
> *"The user wants me to find Python files that import pandas. I have a file_search tool and a file_reader tool. First, I should search for .py files, then read each one to check for pandas imports, then summarize their purposes. Let me start with the search."*

The output of reasoning is a **decision**: which action to take and with what parameters.

Modern LLMs can express this decision as a structured function call:

```
{
  "tool": "file_search",
  "arguments": {
    "pattern": "*.py",
    "path": "./"
  }
}
```

Or they might decide no tool is needed yet and ask a clarifying question:

```
{
  "action": "ask_user",
  "message": "Should I search the entire project or just the src/
directory?"
}
```

1.2.3 3. Act

The agent executes the chosen action. This is where code meets the real world:

- **Tool execution:** Call a function, hit an API, query a database
- **Code execution:** Run a Python script in a sandbox
- **Response generation:** Send a message to the user
- **State modification:** Update a variable, save to a file

In code, this might be as simple as:

```
result = tools[action.tool_name].execute(action.arguments)
```

The runtime controls dispatch, but a tool need not be deterministic. A web API may return different data on a later call, and a network timeout may leave an action's outcome unknown.

1.2.4 4. Observe

The agent processes the result of its action and decides whether to continue or stop. This involves:

- **Interpreting the result:** Did the tool succeed? What did it return?
- **Updating context:** Add the result to the conversation history
- **Checking goal achievement:** Is the task done, or do I need to keep going?
- **Planning next steps:** If not done, what should I do next?

The observation phase closes the loop. The result of the action becomes the next input (perception), and the cycle repeats.

1.2.5 Example: A Research Agent in Action

Let's trace the agent loop through a real task:

User: "What's the latest news about OpenAI?"

Iteration 1: - **Perceive:** User query received - **Reason:** *"I need current news. I have a web_search tool. I'll search for 'OpenAI news'."* - **Act:** `web_search(query="OpenAI news latest")` - **Observe:** Returns 10 search results with titles and URLs

Iteration 2: - **Perceive:** Search results received - **Reason:** *"I have headlines but no details. I should read the top 2 articles to provide a comprehensive answer."* - **Act:** `read_webpage(url="https://techcrunch.com/openai-announces-...")` - **Observe:** Article content retrieved

Iteration 3: - **Perceive:** Article 1 content - **Reason:** *"Got first article. Let me get the second for more context."* - **Act:** `read_webpage(url="https://theverge.com/openai-gpt5-...")` - **Observe:** Article 2 content retrieved

Iteration 4: - **Perceive:** Article 2 content - **Reason:** *"I now have enough information to answer. Let me synthesize the findings."* - **Act:** Generate response: "Based on recent news, OpenAI announced... [summary of both articles with sources]" - **Observe:** Response complete, task finished, exit loop

Notice how each iteration builds on the previous one. The agent doesn't have a predetermined script—it adapts based on what it observes.

1.2.6 The Loop Never Ends (Until It Does)

One critical aspect of agent design is **stopping conditions**. Without them, your agent could loop forever. Common stopping conditions:

1. **Goal achieved:** Required postconditions are verified; generating a final answer alone is not proof
2. **Max iterations reached:** Safety limit (e.g., 50 steps maximum)
3. **Error threshold exceeded:** Too many failed tool calls
4. **User interruption:** User cancels the agent
5. **Resource exhaustion:** Timeout or budget limit hit

In Chapter 4, you'll implement this loop in Python and see it in action.

1.3 A Brief History of AI Agents

Rule-based systems, planning, robotics, and reinforcement learning supplied many of the ideas behind today's agents. LLMs made it practical to interpret a broad range of natural-language tasks without hand-writing a parser for each one; they did not invent the concept of acting on observations.

A few milestones clarify the sequence:

- **2020:** GPT-3 demonstrated broad few-shot language-task performance. Prompting changes context; it does not update model weights. See Brown et al..
- **October 2022:** ReAct combined reasoning traces with actions and observations in experiments. Results depend on the task and model; this was not a universal reliability guarantee. See Yao et al..
- **June 2023:** OpenAI introduced API function calling for GPT-3.5 Turbo and GPT-4. This was not a March 2022 feature. See the release announcement.
- **2023–2024:** Frameworks made tool orchestration, retrieval, and teams easier to experiment with. The underlying loop can still be implemented directly with a provider SDK.
- **November 2024:** Anthropic introduced MCP for connecting applications with tools and data. See the MCP announcement.
- **April 2025:** Google introduced A2A for communication between agent applications. See the A2A announcement.

By the review date of September 6, 2026, model and framework names have changed repeatedly. The durable engineering ideas are explicit control flow, authorized actions, useful observations, bounded resources, and evaluation against real outcomes.

1.4 Real-World Use Cases Across Industries

AI agents aren't science fiction—they're solving real problems today. Here's how different industries are using them:

1.4.1 Software Development

Coding assistants have evolved from autocomplete to autonomous pair programmers: - **coding agents** (2025): You describe a feature in plain English, the agent generates the implementation plan, writes code across multiple files, creates tests, and opens a PR. - **Code review agents**: Analyze pull requests, flag security issues, suggest performance improvements, check for style violations. Unlike static analyzers, they understand context and intent. - **CI/CD agents**: Monitor build failures, automatically bisect to find the breaking commit, propose fixes, and verify them against test suites.

Why it works: Agents can navigate codebases (searching, reading files), use compilers and test runners as tools, and iterate until tests pass. The reasoning capability means they can debug—not just apply templates.

1.4.2 Customer Support

Autonomous ticket resolution is changing support economics: - **L1 agent**: Takes the ticket, searches knowledge bases and past tickets for similar issues, gathers system logs, applies standard fixes. Escalates to human only when necessary. - **Escalation agents**: Route complex tickets to the right human specialist based on semantic understanding of the issue. - **Follow-up agents**: Monitor ticket status, automatically check if fixes worked, close resolved tickets, survey customers.

Organizations can measure the fraction of support requests resolved without escalation; there is no universal resolution rate for agents.

1.4.3 Research and Analysis

Literature review agents compress months of work into hours: - Input: "Summarize the current state of research on quantum error correction in topological qubits" - Agent: Searches academic databases, downloads 50 relevant papers, extracts key findings, identifies trends and gaps, generates a comprehensive report with citations.

Data analysis agents: Connected to databases and data warehouses, they can be asked "Why did sales drop 15% in Q2?" and will: 1. Query sales data, segment by region/product/channel 2. Identify anomalies (Midwest sales down 40%) 3. Cross-reference with external data (regional recession, competitor launched) 4. Generate visualizations and explanations

Why it works: Research is fundamentally about finding, reading, synthesizing, and reasoning over information—tasks that LLMs excel at when augmented with search and retrieval tools.

1.4.4 Finance

Financial analysis agents combine reasoning with real-time data: - **Earnings analysis**: Reads 10-K filings, extracts key metrics, compares to analyst expectations, generates investment thesis. - **Compliance checking**: Scans transactions for suspicious patterns, cross-references against regulations, flags potential violations with explanations. - **Portfolio optimization**: Analyzes market conditions, evaluates risk across holdings, proposes rebalancing strategies.

Why it works: Financial workflows involve structured data (spreadsheets, reports, APIs), clear rules (regulations, accounting standards), and reasoning under uncertainty—a perfect match for agent capabilities.

1.4.5 Healthcare

Clinical decision support agents (heavily regulated, human-in-the-loop): - **Diagnostic assistance**: Given symptoms and test results, suggests possible diagnoses with medical literature citations. - **Treatment planning**: Searches clinical trial databases, identifies relevant protocols, surfaces recent research on medications. - **Medical literature synthesis**: "What's the latest evidence on X for patients with Y condition?" → comprehensive summary with confidence levels.

Why it works: Medicine involves keeping up with an overwhelming volume of research. Agents can retrieve and synthesize information far faster than humans, though final decisions remain with doctors.

1.4.6 Legal

Contract review agents accelerate due diligence: - Upload 200-page merger agreement → agent extracts key terms, identifies unusual clauses, flags risks, compares to standard language, produces summary. - **Legal research agents**: "Find cases where courts ruled on force majeure clauses for pandemics in New York" → retrieves relevant case law, summarizes holdings. - **Document generation**: Creates first drafts of contracts, NDAs, and filings based on templates and case-specific details.

Why it works: Legal work involves reading, cross-referencing, and pattern matching across huge document sets—tasks where agents shine.

1.4.7 Marketing

Content creation pipelines run entire campaigns with agent teams: - **Researcher agent**: Analyzes target audience, searches trending topics, identifies competitor content gaps. - **Writer agent**: Generates blog posts, social media content, email campaigns. - **SEO agent**: Optimizes for keywords, adds metadata, suggests internal links. - **Editor agent**: Reviews for brand voice, factual accuracy, and engagement potential.

Campaign optimization agents: Monitor ad performance, test variations, reallocate budget, generate performance reports.

Why it works: Marketing workflows are repetitive but require creativity and context—agents can handle the repetition while LLMs provide the creative reasoning.

1.4.8 Why Agents Are Transforming These Industries

The pattern is consistent across all these use cases:

1. **Autonomy**: Agents don't need step-by-step instructions—they figure out the steps.
2. **Tool use**: They can interact with existing systems (databases, APIs, filesystems) without custom integrations.
3. **Reasoning**: They adapt to unexpected situations instead of failing.
4. **Scale**: One agent can do the work of many humans for repetitive cognitive tasks.
5. **24/7 availability**: They never sleep, never get tired, retain selected context within storage and retrieval limits.

The tasks being automated aren't just data entry—they're knowledge work that previously *required* human judgment. That's the shift.

But agents aren't replacing humans—they're changing what humans do. Support agents handle the complex emotional cases. Developers focus on architecture and design. Lawyers provide strategic counsel. Agents handle the grunt work.

In the rest of this book, you'll learn to build agents like these—starting with simple tool-using agents and progressing to sophisticated multi-agent systems that can tackle real-world problems.

1.5 Summary

Key Takeaways:

1. **AI agents are autonomous systems** that pursue goals by reasoning about actions, using tools, and adapting based on observations. They differ from chatbots (no actions) and automation scripts (no reasoning).
2. **The agent loop** is the fundamental pattern: Perceive → Reason → Act → Observe → Repeat. Every agent in this book follows this cycle.
3. **LLMs made practical agents possible.** Before 2022, building useful general-purpose agents was impractical. The combination of LLM reasoning + tool use + function calling unlocked the field.
4. **The ReAct pattern** (reasoning + acting) is the foundation of modern agent design. Interleaving thinking with action-taking dramatically improves performance.
5. **Real-world agents are already in production** across industries: code generation, customer support, research synthesis, financial analysis, legal review, and content creation. They work because they combine autonomy, tool use, and reasoning.
6. **Not everything needs to be an agent.** Use chatbots for Q&A, automation scripts for predetermined workflows, and agents when tasks require reasoning and adaptation.

In the next chapter, we'll dive into the core building blocks you need to construct agents: LLMs, prompt engineering, tool use, memory management, and structured output.

1.6 Exercises

1. **Identify automation opportunities**: List 3 tasks you do regularly at work or in personal projects. For each, determine:
   - Could a chatbot handle it? (Does it just require answering questions?)
   - Could a script handle it? (Are the steps predetermined?)
   - Does it need an agent? (Does it require reasoning and adaptation?)
   - What tools would the agent need access to?
2. **Design an agent loop**: Pick one task from above and trace through the agent loop:
   - **Perceive**: What's the initial input?
   - **Reason**: What decision does the agent need to make?
   - **Act**: What action should it take?
   - **Observe**: What result does it get, and does it continue or stop?
   - Draw this as a diagram for at least 2 iterations.
3. **Analyze an existing AI product**: Choose one of these (or find your own):
   - GitHub Copilot
   - ChatGPT with web browsing
   - A customer support chatbot you've interacted with

   Classify it: Is it a chatbot, assistant, or agent? What evidence supports your classification? What capabilities would need to be added to move it to the "agent" category?
4. **Compare approaches**: Take the task "book a flight to New York next month":
   - How would a traditional automation script handle this?
   - How would a chatbot handle this?
   - How would an AI agent handle this?
   - What's the key difference in approach?
5. **Stopping condition design**: An agent is tasked with "debug why the app crashed." It has access to logs, code files, and the ability to run tests. Design 5 different stopping conditions (3 for success, 2 for failure cases) that prevent the agent from running forever.

Reflection question: Which tasks in your domain are most suitable for agents versus other approaches? What makes them good agent tasks?