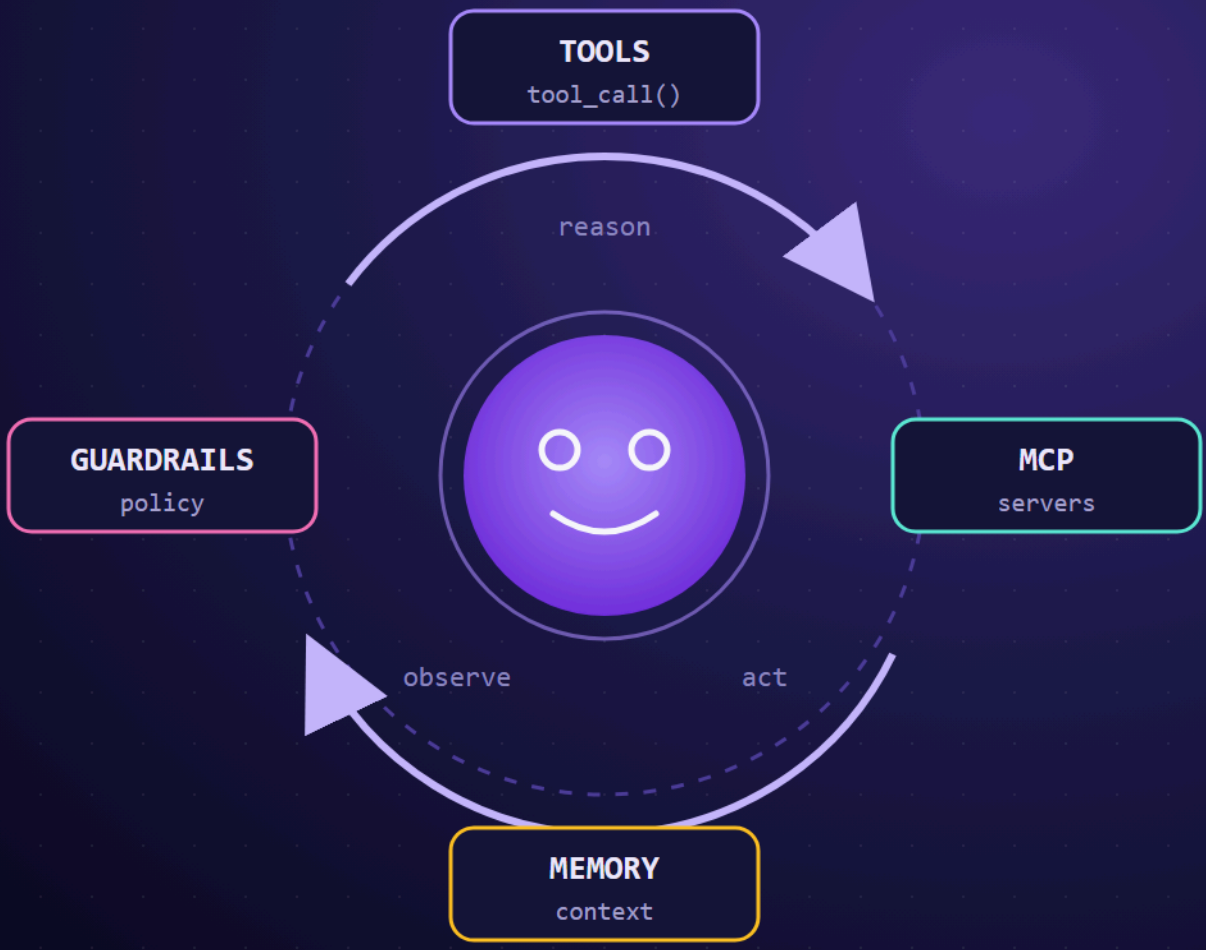


THE PRACTICAL GUIDE

Building AI Agents

Tool Calling, MCP, Memory, and Guardrails

From your first agent loop to safe, observable agents in production

PYTHON · TYPESCRIPT · MCP · RAG · EVALS

THENEXBOOKS

Building AI Agents

A Practical Guide to Tool Calling, MCP, Memory, and Guardrails

From Your First Agent Loop to Safe, Observable Agents in Production

First Edition

THENEXBOOKS

Building AI Agents
A Practical Guide to Tool Calling, MCP, Memory, and Guardrails

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. AI models, provider APIs, SDKs, and protocols such as the Model Context Protocol change quickly; always consult the official documentation for the versions you use. Model names in this book are written as placeholders such as `<chat-model>`; substitute the models available to you.

All product, company, and protocol names mentioned in this book are trademarks or registered trademarks of their respective owners. Their use is for identification only and does not imply endorsement or affiliation.

Keystead, the running example in this book, is fictional. Any resemblance to real companies, products, or people is coincidental.

Preface

A chatbot answers questions. An agent gets things done. Give a language model a goal, a set of tools it can call, and a loop that lets it observe the results of its actions, and it can look up a lease, file a maintenance request, check a vendor's calendar, and report back — without a human writing each step in advance.

That power is exactly what makes agents hard to build well. An agent that can act can also act wrongly: call the wrong tool, pass the wrong argument, loop forever, leak data it should not see, or follow instructions hidden in a document it was only supposed to read. Demos are easy. Agents that are reliable, secure, affordable, and observable in production are not.

This book is a practical guide to building the second kind. We start from first principles — what an agent is and how the agent loop works — and build a minimal agent in plain Python before touching any framework. Then we go deep on the four capabilities that separate toy agents from real ones: **tool calling**, the **Model Context Protocol (MCP)**, **memory**, and **guardrails**. Finally, we cover evaluation, observability, multi-agent systems, and what it takes to run agents in production.

Who This Book Is For

- **Software developers** who have called an LLM API and now want to build agents that use tools and take actions.
- **Backend and platform engineers** responsible for connecting agents to real systems safely.
- **Tech leads and architects** deciding when an agent is the right design, and how to evaluate, secure, and operate one.
- **AI engineers** who want a vendor-neutral, framework-neutral understanding of how agents actually work.

What You Will Learn

- How LLMs power agents, and how to implement the reason–act–observe loop yourself.
- Tool calling in depth: schemas, tool design, structured outputs, validation, errors, retries, and parallel calls.
- The Model Context Protocol: building MCP servers in Python and TypeScript, connecting clients, and running MCP securely.
- Memory and context: token budgets, summarization, retrieval, persistent memory, and planning patterns.
- Guardrails: input, output, and action controls; prompt-injection defenses; and human-in-the-loop approvals.
- Evaluation, tracing, and cost control — and how to ship multi-agent systems to production.

How This Book Is Organized

Part I — Foundations of AI Agents (Chapters 1–5) explains what agents are, how LLMs power them, and how to build a minimal agent. **Part II — Tool Calling** (Chapters 6–10) covers the mechanics and design of tools. **Part III — The Model Context Protocol** (Chapters 11–15) builds and secures MCP servers and clients. **Part IV — Memory, Knowledge, and Planning** (Chapters 16–20) gives agents context, knowledge, and the ability to plan. **Part V — Guardrails, Safety, and Evaluation** (Chapters 21–25) makes agents safe, measurable, and observable. **Part VI — Multi-Agent Systems and Production** (Chapters 26–28) scales up and brings everything together in a complete case study. The appendices provide a cheat sheet, templates, a production checklist, and a glossary.

Each chapter ends with a **Key Takeaways** box for quick review.

The Running Example: Keystead

Throughout the book we build agents for **Keystead**, a fictional property-management platform. Property managers run apartment buildings, residents pay rent and report problems, maintenance vendors fix things, and prospective renters book tours. Keystead's AI team builds four agents: a **Resident Assistant**, a **Maintenance Dispatcher**, a **Leasing Agent**, and an **Ops Analyst**. They share one set of tools and must never move money or approve leases on their own.

Models, Versions, and Conventions

This book is vendor-neutral. Code uses a small wrapper, introduced in Chapter 4, instead of one provider's SDK, and model names appear as placeholders such as `<chat-model>`, `<small-model>`, and `<embedding-model>`. We do not quote prices, context-window sizes, or benchmark results, because they change too quickly; we explain the mechanisms and show you how to measure. Specifications that are still evolving are labeled as such. Examples use Python 3.11 or later, with TypeScript in the MCP chapters.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version details, or a useful aside.

> **WARNING**
> A common mistake, or behavior that causes incidents or runaway costs.

Contents

Preface

PART I — FOUNDATIONS OF AI AGENTS

1 What Is an AI Agent? From Chatbots to Autonomous Systems

2 How LLMs Power Agents: Tokens, Context Windows, and Structured Output

3 The Agent Loop: Reason, Act, Observe

4 Your First Agent in Python: A Minimal Loop Without Frameworks

5 Instructions for Agents: System Prompts, Roles, and Policies

PART II — TOOL CALLING

6 Tool Calling Fundamentals: Schemas, Arguments, and Results

7 Designing Good Tools: Naming, Granularity, and Descriptions

8 Structured Outputs and Validation with JSON Schema and Pydantic

9 When Tools Fail: Errors, Retries, Timeouts, and Idempotency

10 Parallel, Sequential, and Long-Running Tool Calls

PART III — THE MODEL CONTEXT PROTOCOL (MCP)

11 Understanding MCP: Hosts, Clients, Servers, and Messages

12 Building an MCP Server in Python

13 Building an MCP Server in TypeScript and Connecting Clients

14 MCP Resources, Prompts, and Client Features

15 MCP in Production: Transports, Authorization, and Security

PART IV — MEMORY, KNOWLEDGE, AND PLANNING

16 Agent Memory Architecture: Short-Term, Long-Term, and Working Memory

17 Managing Context: Summarization, Compaction, and Token Budgets

18 Retrieval for Agents: Embeddings, Vector Search, and Agentic RAG

19 Persistent Memory Stores: Profiles, Episodes, and Facts

20 Planning and Reasoning Patterns: ReAct, Plan-and-Execute, and Reflection

PART V — GUARDRAILS, SAFETY, AND EVALUATION

21 Guardrails Fundamentals: Input, Output, and Action Controls

22 Prompt Injection and Tool Security

23 Human-in-the-Loop: Approvals, Permissions, and Escalation

24 Evaluating Agents: Test Sets, Trajectory Evals, and LLM-as-Judge

25 Observability, Tracing, and Cost Control

PART VI — MULTI-AGENT SYSTEMS AND PRODUCTION

26 Multi-Agent Systems: Orchestrators, Handoffs, and Agent-to-Agent Protocols

27 Shipping Agents to Production: Architecture, Frameworks, Scaling, and Reliability

28 Case Study: The Keystead Resident and Maintenance Agents End to End

APPENDICES

A AI Agent Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

Foundations of AI Agents

What an agent really is, how language models power it, and how the reason–act–observe loop works — ending with a complete minimal agent built in plain Python and the instructions that steer it.

1 · What Is an AI Agent?
2 · How LLMs Power Agents
3 · The Agent Loop
4 · Your First Agent in Python
5 · Instructions for Agents

CHAPTER 1 · PART I — FOUNDATIONS OF AI AGENTS

What Is an AI Agent? From Chatbots to Autonomous Systems

"Agent" is one of the most overloaded words in software right now. This chapter gives you a precise, working definition — goal, tools, loop, and autonomy — shows how agents differ from chatbots and fixed workflows, explains when you should not build one at all, and introduces Keystead, the property-management platform whose four agents we build throughout the book.

From Answering to Doing

Imagine a resident of an apartment building managed on the Keystead platform. It is late in the evening, the kitchen sink is leaking, and they open the resident app. The first generation of Keystead's assistant was a chatbot: a large language model (LLM) with a system prompt and a copy of the building's FAQ pasted in. It could answer "Are dogs allowed?" reasonably well. But when the resident typed "My sink is leaking, can someone come tomorrow?", the best it could do was reply with a polite paragraph that ended in "Please submit a maintenance request through the app." The resident was already in the app. They had asked for help, and they got instructions.

The gap between those two experiences is the gap this book is about. A chatbot *answers*. An agent *does*: it can look up the resident's unit, check whether a similar work order is already open, create a new one with the right urgency and trade, and tell the resident what happens next. It does this not because someone wrote an `if "leak" in message` branch, but because a model was given a goal, a set of tools it can call, and a loop in which it can decide what to do next based on what it has observed so far.

That last sentence contains almost everything you need to know about agents. The rest of this chapter unpacks it, and the rest of the book shows you how to build it so that it is reliable, safe, and affordable enough to put in front of real people.

A Working Definition

There is no single standard definition of "AI agent", and vendors use the word to mean anything from a chatbot with a search button to a fully autonomous background worker. For this book we use a practical, engineering-oriented definition:

> **NOTE**
>
> An **AI agent** is a system in which a language model, given a goal, repeatedly chooses actions — usually calls to tools — observes their results, and decides what to do next, until the goal is met or a stopping condition is reached. The model directs the control flow; the surrounding code executes actions and enforces limits.

Four ingredients appear in that definition, and each deserves a closer look.

1. A goal

An agent works toward an outcome, not just a reply. "Answer this question" is a goal, but so is "get this leak fixed", "find a tour slot that works for this prospect", or "summarize last week's overdue work orders for the manager of Birch Court". Goals come from two places: the user's request in the current conversation, and the standing instructions the developer gives the agent in its system prompt (the subject of Chapter 5). A goal needs a recognizable finish line. If you cannot say what "done" looks like, the agent cannot either, and it will either stop too early or wander.

2. Tools

A tool is a function the model can ask your code to run. The model never executes anything itself; it produces a structured request — a tool name plus arguments — and your code decides whether and how to carry it out. Tools are how an agent perceives the world beyond its training data (reading a lease, searching the house-rules knowledge base) and how it changes the world (creating a work order, scheduling a tour). Here is what a tool definition looks like when you hand it to a model. Every major provider uses essentially this shape: a name, a natural-language description, and a JSON Schema describing the arguments.

keystead/tools/create_work_order.json

```
{
  "name": "create_work_order",
  "description": "File a work order for a unit after confirming the problem and urgency.",
  "input_schema": {
    "type": "object",
    "properties": {
      "unit_id": {"type": "string", "description": "Keystead unit ID, e.g. U-1042"},
      "category": {
        "type": "string",
        "enum": ["plumbing", "electrical", "hvac", "appliance", "other"]
      },
      "urgency": {"type": "string", "enum": ["emergency", "urgent", "routine"]},
      "summary": {"type": "string", "maxLength": 500},
      "permission_to_enter": {"type": "boolean"}
    },
    "required": ["unit_id", "category", "urgency", "summary"]
  }
}
```

The description is the part most teams underinvest in. The model reads it every time it decides whether to call the tool, so it is effectively a piece of the prompt. The schema, meanwhile, constrains the *shape* of the arguments — an `enum` for urgency means the model cannot invent a fourth level — but it cannot guarantee the *values* are right. The model can still pick the wrong unit. Chapter 6 covers tool schemas in depth, and Chapter 7 is entirely about writing names and descriptions so that models use tools correctly.

3. A loop

A single model call can request a tool, but it cannot see the result of that tool in the same call. To act on results, the system has to call the model again with the tool's output appended to the conversation. Repeat that and you have the **agent loop**: the model reasons about the situation, chooses an action, your code performs it, the result is fed back as an observation, and the cycle continues. The loop is what lets an agent recover when the first lookup returns nothing, ask a clarifying question, or chain several steps together without you having written the sequence in advance. Chapter 3 is devoted to the loop; Chapter 4 builds one in a few hundred lines of plain Python.



4. Autonomy: who decides the next step?

The defining property of an agent is not that it uses an LLM, or even that it calls tools. It is that **the model decides the control flow**. In ordinary software, the programmer decides which function runs next. In an agent, the programmer decides which functions are *available*, and the model picks among them at runtime. That shift is the source of both the power and the risk of agents. It lets a system handle requests nobody anticipated, and it lets a system take actions nobody anticipated. Most of the engineering in this book — tool design, validation, guardrails, human approval, evaluation — exists to capture the first benefit while containing the second risk.

Chatbots, Workflows, and Agents

It helps to place agents on a spectrum alongside two things you have probably already built: chatbots and LLM workflows.

A **chatbot** is a model in a conversation. It takes messages in and produces messages out. It may be very good at language, and it may have documents pasted into its context, but it does not act on external systems, and its "control flow" is simply one call per user turn.

An **LLM workflow** (sometimes called a pipeline or chain) uses one or more model calls inside a control flow that *you* wrote. For example: classify the incoming message, then, if it is a maintenance request, extract the fields with a second call, then create the work order with ordinary code. Each step may use an LLM, but the sequence is fixed and visible in the source.

An **agent** hands the sequencing to the model. You give it tools and a goal; it decides which tools to call, in which order, and when it is done.

Property	Chatbot	Workflow	Agent
Who picks next step	Nobody (one reply)	Your code	The model
Acts on systems	No	Yes, fixed steps	Yes, chosen steps
Handles the unexpected	In words only	Poorly	Often well
Predictability	Medium	High	Lower
Testing	Output checks	Unit tests per step	Trajectory evals
Cost per request	One call	Known number	Variable

The difference is easiest to see in code. Here is a workflow that handles maintenance messages. Every branch is explicit. The LLM is used twice, as a classifier and as an extractor, but it never decides what happens next.

keystead/workflows/maintenance_intake.py

```
from keystead import llm
from keystead.api import keystead_api


def handle_resident_message(resident_id: str, text: str) -> str:
    """A fixed LLM workflow: classify, extract, act. The code owns the flow."""
    label = llm.classify(
        text, labels=["maintenance", "house_rules", "billing", "other"]
    )
    if label != "maintenance":
        return route_elsewhere(label, resident_id, text)

    fields = llm.extract(
        text,
        schema={
            "category": ["plumbing", "electrical", "hvac", "appliance", "other"],
            "urgency": ["emergency", "urgent", "routine"],
            "summary": "string",
        },
    )
```

```
    unit_id = keystead_api.get_resident(resident_id)["unit_id"]
    order = keystead_api.create_work_order(unit_id=unit_id, **fields)
    return f"Thanks! Work order {order['id']} is open. We'll be in touch."
```

And here is the shape of the agent version. The branching has disappeared from the code; it now lives in the model's choices. (We use `llm.classify` and `llm.extract` above only as illustrative helpers; the real wrapper we build in Chapter 4 exposes a single `llm.chat` function.)

keystead/agents/sketch.py

```
from keystead import llm
from keystead.tools import registry

SYSTEM = "You are Keystead's Resident Assistant. Help residents of their building."


def handle_resident_message(resident_id: str, text: str, max_steps: int = 8) -> str:
    """An agent: the model decides which tools to call and when to stop."""
    messages = [
        {"role": "system", "content": SYSTEM},
        {"role": "user", "content": f"[resident {resident_id}] {text}"},
    ]
    for _ in range(max_steps):
        reply = llm.chat(messages, tools=registry.schemas())
        messages.append(reply.as_message())
        if not reply.tool_calls:
            return reply.text                  # the model chose to stop
        for call in reply.tool_calls:
            result = registry.execute(call.name, call.arguments)
            messages.append(
                {"role": "tool", "tool_call_id": call.id, "content": result}
            )
    return "Sorry, I couldn't finish that. A team member will follow up."
```

Notice what the agent version gains. If the resident writes "the sink is leaking again — is that the same ticket as last week?", the workflow has no branch for that; the agent can call `get_work_order` first. If the resident also asks whether they are allowed to use a portable dishwasher, the agent can call `search_knowledge_base` in the same conversation. Notice also what it loses: you can no longer read the source and know exactly which calls will happen. That loss is real, and it is why the next section exists.

> **TIP**
>
> The spectrum is not a ladder you must climb. Many of the best production systems are workflows with one agentic step inside them — for instance, a fixed intake pipeline that hands only the ambiguous messages to a small agent. Start at the simplest point on the spectrum that solves the problem, and move right only when you can name the requests the simpler design fails on.

Levels of Autonomy

"Autonomous" is not binary. It is more useful to think about how much the system is allowed to do without a human confirming it. The following levels are not an industry standard, but we use them throughout the book as a shared vocabulary.

Level	Name	What the agent may do
0	Assistant	Answer and draft; no tools that act
1	Read-only agent	Call lookup tools; humans act
2	Propose-and-confirm	Prepare actions; human approves each
3	Bounded actor	Take low-risk actions alone
4	Background worker	Run unattended within policy

Autonomy is best assigned *per tool*, not per agent. Keystead's Resident Assistant is a Level 3 agent for `create_work_order` — filing a routine repair request is low risk and easily reversed — but a Level 2 agent for `send_resident_message`, which sends a message that a human must approve first. Some actions sit outside the scale entirely: no Keystead agent may move money, change rent, or approve a lease, at any level. Those actions are simply not exposed as tools. The safest tool is the one the model does not have.

Thinking per tool also clarifies what "raising autonomy" means over a product's life. Teams typically launch at a lower level, collect traces of what the agent proposed and what humans approved or corrected, and then promote individual tools to a higher level once the evidence supports it. Chapter 23 covers approval flows and permission models; Chapter 24 covers the evaluations that give you that evidence.

When Not to Use an Agent

Agents are exciting, and that is exactly why teams reach for them when something simpler would be better. Before you build one, run through the following questions honestly.

Is the sequence of steps already known?

If you can draw the flowchart and it rarely changes, write it as code. A monthly rent-reminder job that looks up residents with a balance due and sends a templated message does not need a model deciding what to do next. It needs a scheduled job. Adding an agent makes it slower, more expensive, harder to test, and capable of surprising you.

Does a single model call suffice?

Many "agent" requests are really extraction, classification, summarization, or drafting tasks. Summarizing a lease into plain language, tagging incoming messages by topic, or turning a voicemail transcript into a structured work-order draft are all single calls with structured output (Chapter 2 and Chapter 8). If there is nothing to observe and react to, there is no reason for a loop.

Can you tolerate variable cost and latency?

An agent's cost is the sum of every model call in its loop, and each call re-sends the growing conversation. A request that takes one step today may take five tomorrow because a tool returned something unexpected. If you have a hard latency budget — say, a synchronous API that must respond within a fixed window — or a strict per-request cost ceiling, an open-ended loop is a poor fit unless you cap it tightly. Measure on your own traffic; do not assume.

What is the cost of a wrong action?

Models make mistakes, and agents turn mistakes into actions. Reading the wrong FAQ entry is cheap. Scheduling a vendor into the wrong unit is expensive and embarrassing. Moving money is unacceptable. The higher the cost of a wrong action and the harder it is to reverse, the more you should push toward workflows, read-only tools, or human approval.

Can you evaluate it?

If you cannot describe what a good outcome looks like well enough to write test cases, you cannot tell whether your agent is getting better or worse when you change a prompt or a model. "It seemed fine in the demo" is not an evaluation strategy. Chapter 24 shows how to build test sets and trajectory evaluations; if you cannot imagine writing one for your use case, pause.

> **WARNING**
>
> A common failure pattern is the "agent-shaped workflow": a system with a known, fixed sequence of steps that was built as an agent anyway. It works in demos, then in production it occasionally skips a step, repeats a step, or invents a step, and nobody can reproduce why. If your agent's traces almost always show the same tool sequence, that is a strong signal to rewrite it as a workflow.

A lightweight way to make this decision explicit is to write it down. The following helper is deliberately simple — it is a thinking aid you can drop into a design document, not a scoring model — but it forces the team to answer the questions above before writing any agent code.

docs/should_this_be_an_agent.py

```
from dataclasses import dataclass


@dataclass
```

```
class UseCase:
    name: str
    steps_known_in_advance: bool
    needs_multiple_observations: bool
    worst_action_reversible: bool
    has_latency_or_cost_ceiling: bool
    can_write_eval_cases: bool


def recommend(case: UseCase) -> str:
    if not case.can_write_eval_cases:
        return "Stop: define success and test cases first."
    if case.steps_known_in_advance:
        return "Workflow: write the sequence as code; use LLM calls inside steps."
    if not case.needs_multiple_observations:
        return "Single call: structured output is enough."
    if not case.worst_action_reversible:
        return "Agent with approvals: read-only tools plus human-confirmed actions."
    if case.has_latency_or_cost_ceiling:
        return "Bounded agent: small tool set, strict max_steps, fallbacks."
    return "Agent: a good fit. Start with read-only tools and add actions gradually."


print(recommend(UseCase(
    name="Monthly rent reminder",
    steps_known_in_advance=True,
    needs_multiple_observations=False,
    worst_action_reversible=True,
    has_latency_or_cost_ceiling=False,
    can_write_eval_cases=True,
)))
# Workflow: write the sequence as code; use LLM calls inside steps.
```

Meet Keystead

Abstract definitions only get you so far. To keep the book concrete, every chapter builds on the same fictional company. **Keystead** is a property-management platform. Property managers use it to run apartment buildings. Residents use it to pay rent, report problems, and ask questions. Maintenance vendors — plumbers, electricians, HVAC technicians — receive and complete jobs through it. Prospective renters use it to find available units and book tours.

Keystead already has a conventional REST API at `api.keystead.example`. The agents we build never talk to the database directly; they call this API through tools, which means the API's existing authorization, validation, and audit logging still apply. That is a design choice worth copying: **an agent should be just another client of your existing systems**, not a privileged backdoor into them.

The domain model

The entities you will meet throughout the book are summarized below. Payments appear in the list because agents need to *read* balances, but no agent can create, modify, or refund a payment.

Entity	What it represents	Agent access
Property	A building or complex	Read
Unit	An apartment in a property	Read
Resident	A tenant living in a unit	Read
Lease	Terms between resident and owner	Read summary
WorkOrder	A maintenance job	Read, create
Vendor	A contractor for a trade	Read
VendorSlot	A vendor's open time window	Read, book
Tour	A prospect's unit viewing	Read, create
Payment	Rent and fee transactions	Read only
Message	Resident communication	Draft; human sends

There is also a **knowledge base** of house-rules and FAQ documents per property: pet policies, quiet hours, parking, package rooms, move-in procedures, and so on. Chapter 18 turns it into a retrieval system; until then, you can think of `search_knowledge_base` as a function that returns the most relevant passages for a query.

The tools

Keystead's AI team has agreed on a single set of tool names, which you will see in code across the book. Keeping names stable matters more than it sounds: evaluation sets, logs, dashboards, and approval policies all refer to tools by name.

Tool	Purpose	Risk
`search_knowledge_base`	Find house-rules and FAQ passages	Low
`get_resident_profile`	Resident's unit and contact prefs	Low (PII)
`get_lease_summary`	Key lease terms and dates	Low (PII)
`get_balance`	Current balance, read-only	Low (PII)
`create_work_order`	Open a maintenance job	Medium
`get_work_order`	Status of an existing job	Low
`list_vendor_slots`	Open vendor time windows	Low
`schedule_vendor_visit`	Book a vendor into a slot	Medium
`list_available_units`	Units open for leasing	Low
`schedule_tour`	Book a prospect's tour	Medium

`send_resident_message`	Message a resident	High: approval
`escalate_to_human`	Hand off to staff	Low

Notice what is missing: there is no `refund_payment`, no `update_rent`, and no `approve_lease`. Keystead's policy is that agents never move money, change rent, or approve leases. Those decisions carry legal and financial consequences, and they stay with humans. When a resident asks for something in that category, the right move is `escalate_to_human`.

Agent 1: the Resident Assistant

The Resident Assistant is the agent residents talk to in the app. It answers questions about house rules, lease terms, and amenities; checks balances; and files maintenance requests. Its tools are `search_knowledge_base`, `get_resident_profile`, `get_lease_summary`, `get_balance`, `create_work_order`, `get_work_order`, and `escalate_to_human`. It is the first agent we build, in Chapter 4, and the one we return to most often. A typical exchange looks like this:

```
Resident:  Are dishwashers allowed? And my bathroom fan stopped working.
Assistant: [calls search_knowledge_base(query="portable dishwasher", property_id="P-7")]
Assistant: [calls get_resident_profile(resident_id="R-3318")]
Assistant: [calls create_work_order(unit_id="U-1042", category="electrical",
            urgency="routine", summary="Bathroom exhaust fan not working")]
Assistant: Portable dishwashers are allowed at Birch Court if they connect to the
           kitchen faucet with the adapter described in section 4.2 of the house
           rules. I've also opened work order WO-88121 for the bathroom fan; a
           technician will contact you to schedule a visit.
```

Agent 2: the Maintenance Dispatcher

The Maintenance Dispatcher works behind the scenes on work orders. It triages new orders — how urgent is this, which trade is needed, does the resident allow entry — looks up vendor availability with `list_vendor_slots`, proposes or books visits with `schedule_vendor_visit`, and drafts updates to residents with `send_resident_message`, which a staff member approves before anything is sent. It is a good example of an agent that is triggered by events rather than chat messages, and it features heavily in Part II on tool calling and in Chapter 28.

Agent 3: the Leasing Agent

The Leasing Agent talks to prospective renters — people who are not yet Keystead users and whose identity is unverified. It answers questions about properties and amenities, checks `list_available_units`, and books tours with `schedule_tour`. Because it faces anonymous members of the public, it is the agent most exposed to adversarial input, and we use it heavily in Chapter 22 on prompt injection. It cannot see any resident data at all.

Agent 4: the Ops Analyst

The Ops Analyst is a back-office agent for property managers. Asked "What's driving the maintenance backlog at Birch Court this month?", it reads work orders and occupancy data and produces a summary. It is read-only by design, which makes it an ideal place to discuss long-running tasks, context management, and planning patterns in Parts II and IV without worrying about side effects.

> **NOTE**
>
> The four agents deliberately span the autonomy scale. The Ops Analyst is Level 1 (read-only). The Leasing Agent and Resident Assistant are Level 3 for a few low-risk actions. The Maintenance Dispatcher mixes Level 3 scheduling with Level 2 messaging. Seeing the same patterns applied at different risk levels is more instructive than any single example.

The Anatomy of an Agent System

A model and a loop make a demo. A production agent is a system with several more parts, and this book is organized around them. It is worth previewing them now so that you can see where each chapter fits.

- **The model.** The language model that reasons and chooses actions. We refer to it with placeholders such as `<chat-model>` and `<small-model>` rather than naming products, because model line-ups change faster than books do. Chapter 2 explains what you need to know about how models behave.
- **Instructions.** The system prompt, role definition, and policies that shape the agent's behavior (Chapter 5).
- **Tools.** The functions the agent can call, their schemas, and how errors and retries are handled (Part II, Chapters 6–10).
- **Integration protocol.** How tools are packaged and shared across applications. The Model Context Protocol (MCP), an open protocol introduced by Anthropic in November 2024, standardizes this, and Part III covers it in five chapters.
- **Memory and knowledge.** What the agent remembers within a conversation and across conversations, and how it retrieves documents (Part IV).
- **Guardrails.** Input checks, output checks, action controls, prompt-injection defenses, and human approvals (Part V).
- **Evaluation and observability.** Test sets, trajectory evaluation, tracing, and cost control, so that you know what the agent is actually doing (Chapters 24 and 25).
- **The runtime.** The loop itself, state management, concurrency, deployment, and multi-agent coordination (Chapters 3, 4, 26, and 27).

It is tempting to think of the model as the agent and everything else as plumbing. In practice the reverse is closer to the truth. The model is a component you rent and swap; the tools, instructions, guardrails, and evaluations are what you own, and they determine whether the system is trustworthy.

What Goes Wrong in Production

Before we start building, it is worth naming the failure modes you will spend the rest of the book defending against. None of them are exotic. Every team that ships agents meets them.

- **Wrong tool, right intent.** The resident asks about a noisy neighbor and the agent files a maintenance work order instead of searching the quiet-hours policy. Usually a tool-description or instruction problem.
- **Right tool, wrong arguments.** The agent creates a work order for the wrong unit because it guessed the unit ID instead of looking it up. Schemas cannot catch this; validation against real data can.
- **Hallucinated facts.** The agent states a pet policy that does not exist because the knowledge-base search returned nothing and it filled the gap. Grounding and "say you don't know" instructions help; evaluation catches what they miss.
- **Loops that do not end.** The agent keeps calling `list_vendor_slots` with slightly different dates, never satisfied. Every loop needs a step limit and a sensible fallback.
- **Runaway cost.** Long conversations and large tool results make each model call more expensive than the last. Context management and budgets are essential (Chapters 17 and 25).
- **Injected instructions.** A work-order note, an email, or a web page contains text that tries to redirect the agent. The model cannot reliably tell data from instructions, so your system must (Chapter 22).
- **Silent drift.** A provider updates a model, or someone edits a prompt, and behavior changes in ways nobody notices until residents complain. Regression evaluations and tracing are how you notice first.

The common thread is that the model is probabilistic and the world is not forgiving. The engineering discipline of agent building is largely about putting deterministic structure — schemas, validators, permissions, limits, tests — around a non-deterministic core.

How to Read This Book

The chapters build on each other, but you can take different paths. If you are new to building on LLMs, read Part I in order; Chapter 2 gives you the mental model of tokens, context windows, and sampling that later chapters assume, and Chapters 3 and 4 give you a working agent you can extend. If you have already built a tool-calling prototype and are fighting reliability issues, skim Part I and go straight to Part II and Part V. If your immediate task is integrating with MCP, Part III stands mostly on its own once you have read Chapter 6.