BUILD THE APPLICATION. VERIFY THE BOUNDARIES.

FULL STACK GENERATIVE & AGENTIC AI WITH PYTHON



500 PAGES • BUILD, TEST, SHIP

INDEPENDENT SELF-STUDY / NOVEMBER 2026

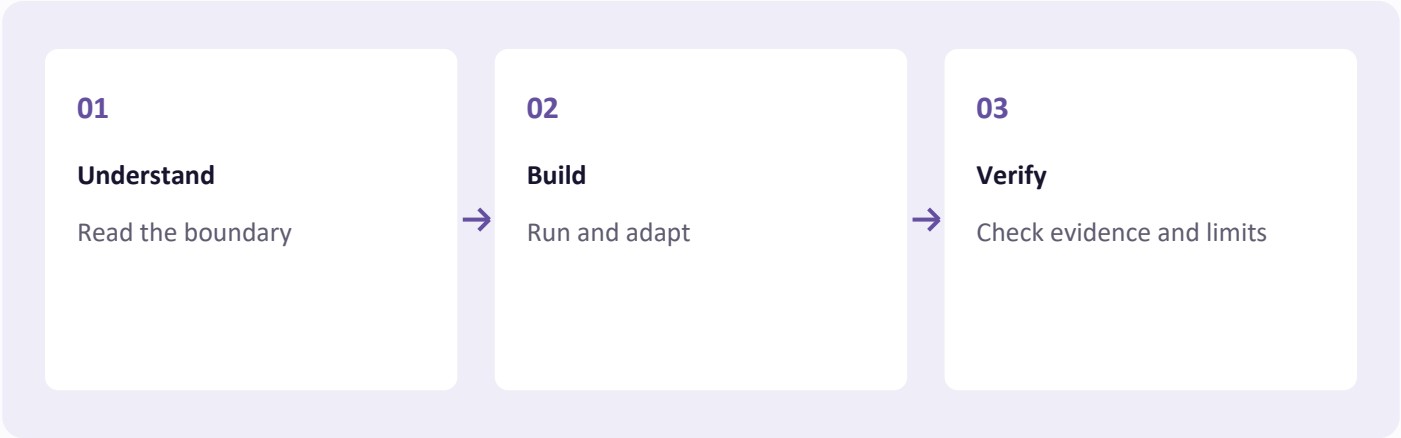
Independent learning edition

Full Stack Generative & Agentic AI with Python

Copyright 2026 Dev. Rarasputin. Independent self-study publication. Not affiliated with Python, FastAPI, OpenAI, a university, or an accreditation body. No certificate, employment outcome, production readiness, or security guarantee is promised.

The guide has exactly 500 pages: twelve opening pages, sixty eight-page chapters, and eight reference/planning pages. It includes code, architecture diagrams, worked fixtures, practice briefs, troubleshooting, transfer exercises, and writing space. It is a practical lab guide, not 500 pages of continuous prose.

You may adapt supplied code and templates for personal and commercial projects. Purchase does not grant resale or redistribution rights to the ebook, artwork, or complete bundle. Third-party packages retain their own licenses. The cover is AI-generated; all teaching diagrams are original vector graphics. No other generated illustration was needed for the technical content.

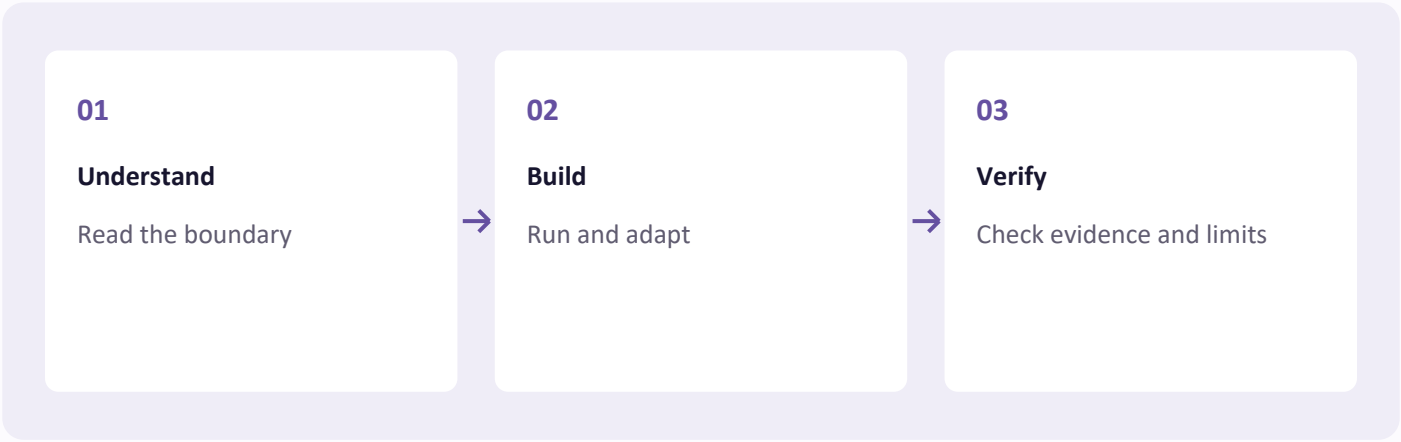


Who this guide is for

You can write small Python functions and use a terminal. You do not need prior agent, retrieval, or web-framework experience. If variables, dictionaries, exceptions, imports, and virtual environments are new, first work through the Python tutorial linked in the references. This is an application-engineering guide rather than a mathematics course or training-from-scratch text.

Work in order for the first four units. Then complete retrieval and tools before durable workflows and evaluation. The final six chapters connect the pieces to Studio, the downloadable local project.

Read each chapter as a loop: understand the boundary, inspect the decision, run the demonstration, attempt the lab, compare the source, verify the checks, diagnose a failure, and transfer the idea. Do not copy a solution into a public service without reviewing its stated limitations.

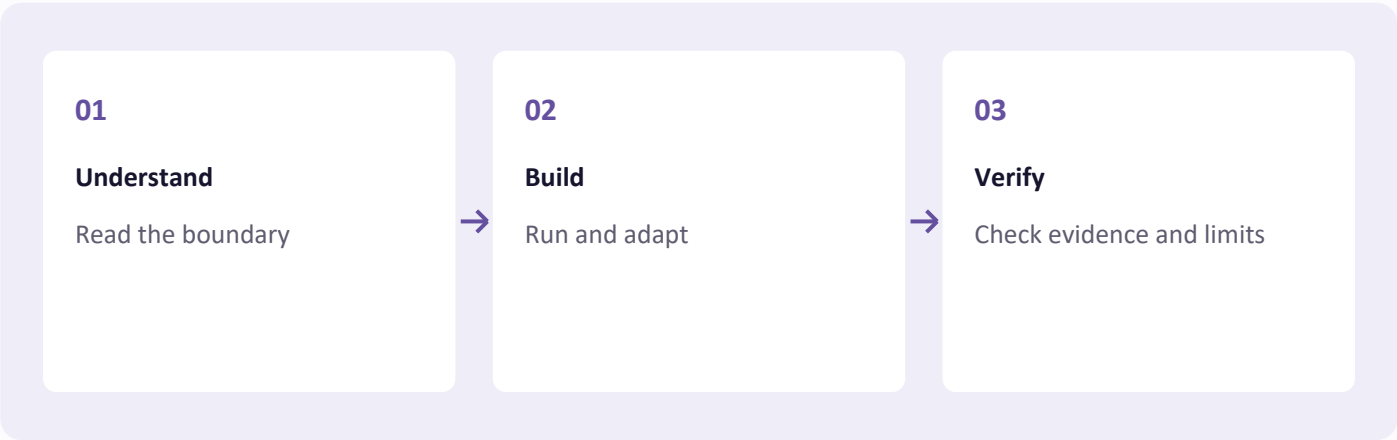


What you will build

Studio is a browser interface backed by FastAPI and SQLite. It searches a fictional handbook with deterministic lexical overlap and returns an extractive offline answer. It can draft a note from evidence, show an exact preview, and save that note only after an explicit commit. The default app makes no model request.

This separation lets you test full-stack behavior without paid API access. The book explains live generation, structured output, function calling, embeddings, streaming, queues, and deployment as separate interfaces and extension exercises. A text-only live Responses adapter is included, opt-in and fake-transport tested. It is not connected to the default app.

No learned embedding model, live model evaluation, autonomous external operator, hosted AI backend, or executed container deployment is claimed. The companion website is a static sales and preview site. The independent project is downloaded and run locally.

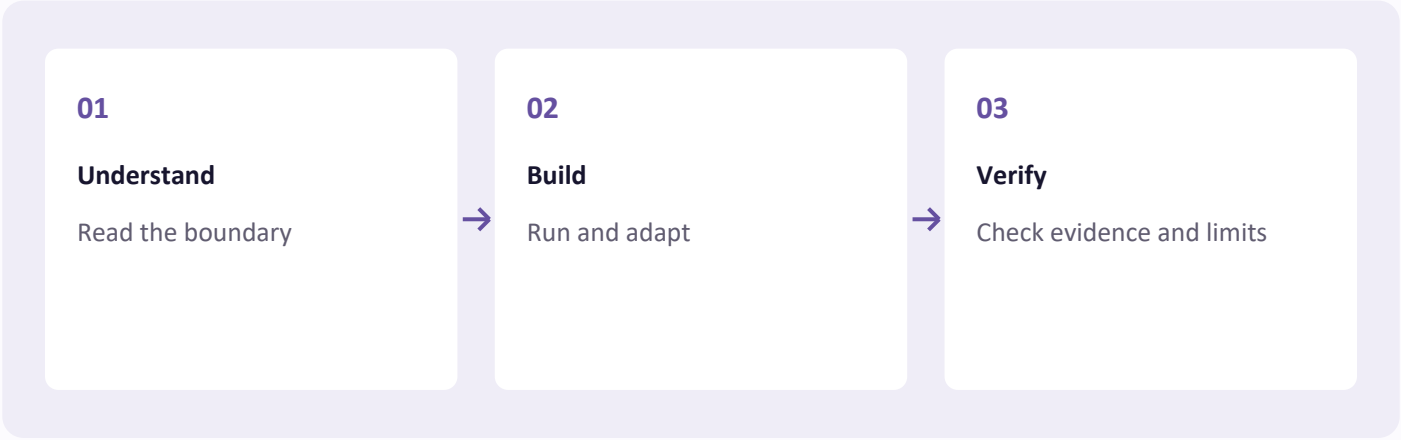


Set up a clean learning environment

Extract the complete ZIP. Read READ-ME-FIRST.txt. Extract AI-Code.zip for chapter programs and Studio-Project.zip for the full-stack project. Keep these folders separate. No real credential is included.

Chapter path: run python verify_bundle.py from the code folder. This uses the standard library. Each example contains its own complete source, and each solution file contains the reference definitions. checks.json contains trusted executable test fixtures distributed with this course; do not replace it with an untrusted file. Run without Python -O so assertions remain enabled.

Project path: follow project/README.md, create a virtual environment, install requirements.txt, set a disposable local STUDIO_TOKEN, run unittest, then start Uvicorn on 127.0.0.1. Dependency installation needs internet access. Offline means the default runtime does not call a model, not that package installation requires no network.

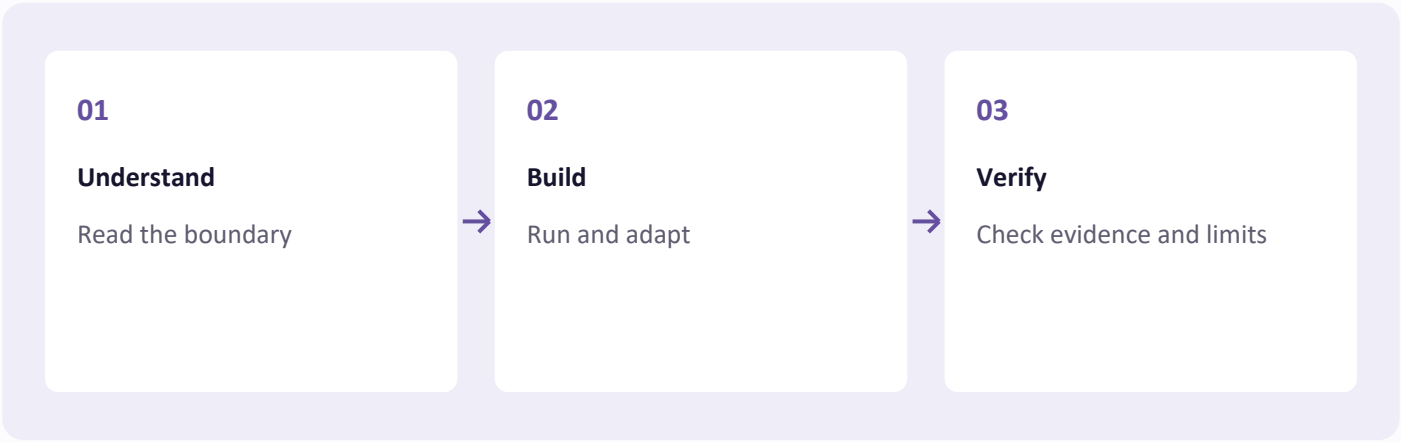


Verification and its limits

All 120 chapter example/solution programs and 180 verification blocks passed on CPython 3.11.9 and 3.12.14 on Windows. The complete FastAPI project has nineteen unittest methods, including eight retrieval expectations and fake live transport checks. It was executed on CPython 3.12.14 with the dependency lock in the download.

The packaged sources are extracted and checked again before release. Browser QA covers supported and unsupported questions, source display, wrong-token errors, note preview and commit, and narrow-screen layout. Tests and screenshots cover the stated fixtures, not arbitrary workloads.

No paid provider request, live function-calling evaluation, learned embedding, cloud model-serving deployment, container run, load benchmark, or production audit was performed. Use a maintained interpreter patch release. Recheck current official documentation when changing dependencies or enabling live access.



A map of the ten units

01 Python and system boundaries: records, validation, configuration, injection, async.

02 The Python web backend: routes, schemas, errors, SQLite, history.

03 A browser interface: forms, fetch, state, rendering, cancellation, streaming protocol.

04 Generative model integration: budgets, prompts, examples, structured output, response adapters.

05 Retrieval and grounding: chunks, lexical search, numeric vector concepts, fusion, evidence packing.

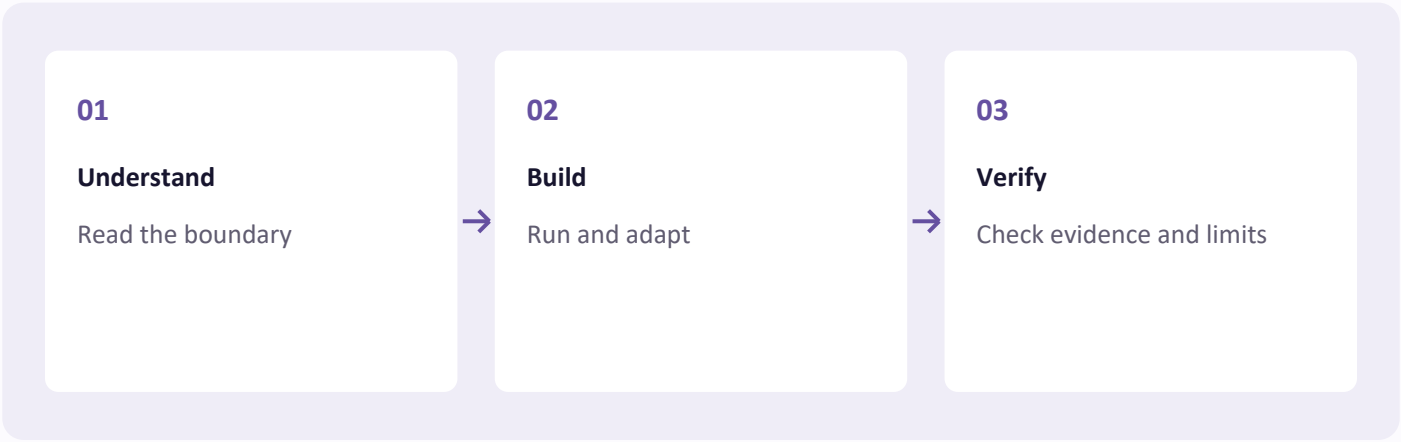
06 Tools and bounded agents: registries, arguments, observations, loops, budgets, exact approval.

07 Durable workflows: routing, state, checkpoints, retries, concurrency, multiple roles.

08 Evaluation and safety: doubles, cases, retrieval metrics, answer contracts, injection, permissions.

09 Operating the system: cache design, cost, traces, latency, queues, readiness.

10 Ship the project: service, browser, corpus, approvals, deployment plan, reproducible release.



Contents / chapters 01-20

01	Map the complete request path	13
02	Use explicit Python records	21
03	Validate at the edge	29
04	Keep configuration on the server	37
05	Define a replaceable provider boundary	45
06	Understand async without guessing	53
07	Treat HTTP as a contract	61
08	Build the first FastAPI route	69
09	Separate request and response schemas	77
10	Design stable error categories	85
11	Persist records with SQLite parameters	93
12	Bound conversation history	101
13	Make the interface contract visible	109
14	Handle fetch success and failure	117
15	Use a small UI state machine	125
16	Render generated text as text	133
17	Handle cancellation and late responses	141
18	Understand streaming event boundaries	149
19	Budget context without pretending to tokenize	157
20	Write prompts as versioned interfaces	165

Contents / chapters 21-40

21	Choose examples that teach the contract	173
22	Validate structured model output	181
23	Normalize provider responses	189
24	Build a live Responses request deliberately	197
25	Chunk documents with provenance	205
26	Build a transparent lexical baseline	213
27	Understand vectors with a small numeric fixture	221
28	Combine rankings without mixing scales	229
29	Pack evidence and validate citations	237
30	Implement a grounded offline answer path	245
31	Build a narrow tool registry	253
32	Parse tool arguments before execution	261
33	Design tool results as evidence	269
34	Run a bounded agent loop	277
35	Track more than one agent budget	285
36	Bind approval to the exact action	293
37	Route tasks with an explicit policy	301
38	Represent workflow state explicitly	309
39	Checkpoint with version checks	317
40	Retry only the failures you intend	325

Contents / chapters 41-60

41	Limit parallel work	333
42	Know when multiple agents add value	341
43	Use test doubles with honest claims	349
44	Build an evaluation case set	357
45	Measure retrieval separately	365
46	Evaluate answers and abstention	373
47	Test prompt-injection boundaries	381
48	Apply permissions before retrieval	389
49	Cache with scope and version	397
50	Estimate cost with labeled assumptions	405
51	Trace behavior without copying private content	413
52	Measure latency distributions	421
53	Use queues for work that outlives a request	429
54	Define readiness before public deployment	437
55	Assemble the service layer	445
56	Connect the browser to the Python app	453
57	Ship the retrieval project	461
58	Ship the approval workflow project	469
59	Plan deployment as a separate implementation	477
60	Release with reproducible evidence	485

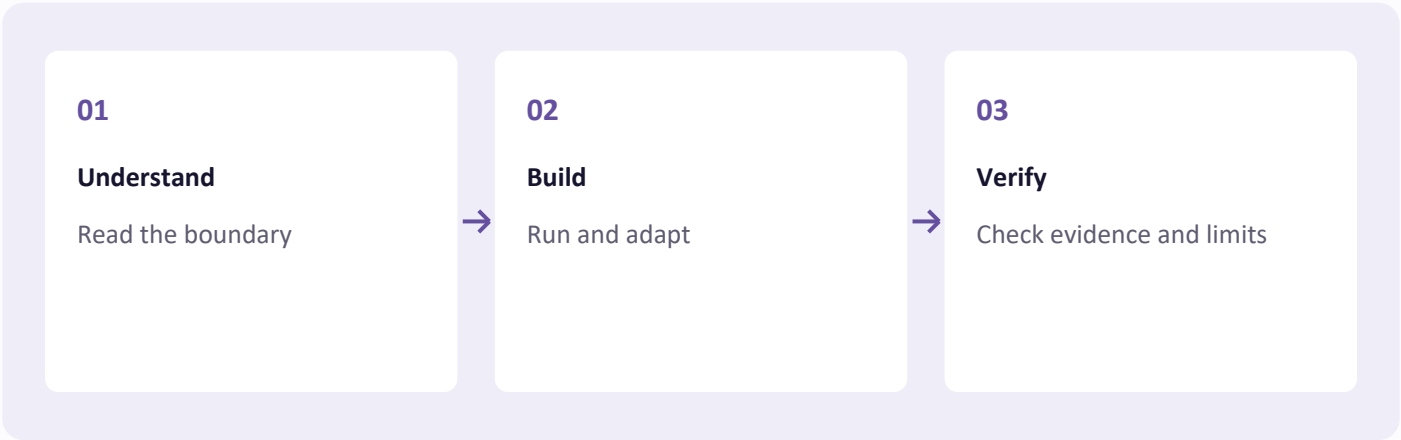
Read the diagrams and the code

The diagrams describe application boundaries and data flow. They are conceptual models, not measured performance or screenshots of an external service. A label such as model proposal means untrusted candidate data until application code validates it. An arrow does not imply permission.

Code pages use monospace text and preserve indentation. The demonstration page shows a driver and observed stdout. The reference page contains the complete function or class definitions. Full runnable files are in the code ZIP; avoid copying code from a reduced website preview.

The three verification blocks in each chapter include valid and rejected behavior. Some blocks contain several assertions or setup statements. The distributed runner executes the full fixtures. Selected comments explain what a test proves and what remains outside its scope.

Writing spaces are for contracts, failure hypotheses, and transfer plans. They are part of the guided lab format; use a separate notebook if you prefer to keep the PDF unmarked.



Working vocabulary

Generative model: a model that produces candidate content conditioned on input. Its output requires application interpretation.

Agent: a system that selects actions over one or more steps using observations. In this guide, deterministic fixtures teach loop mechanics before any optional live model integration.

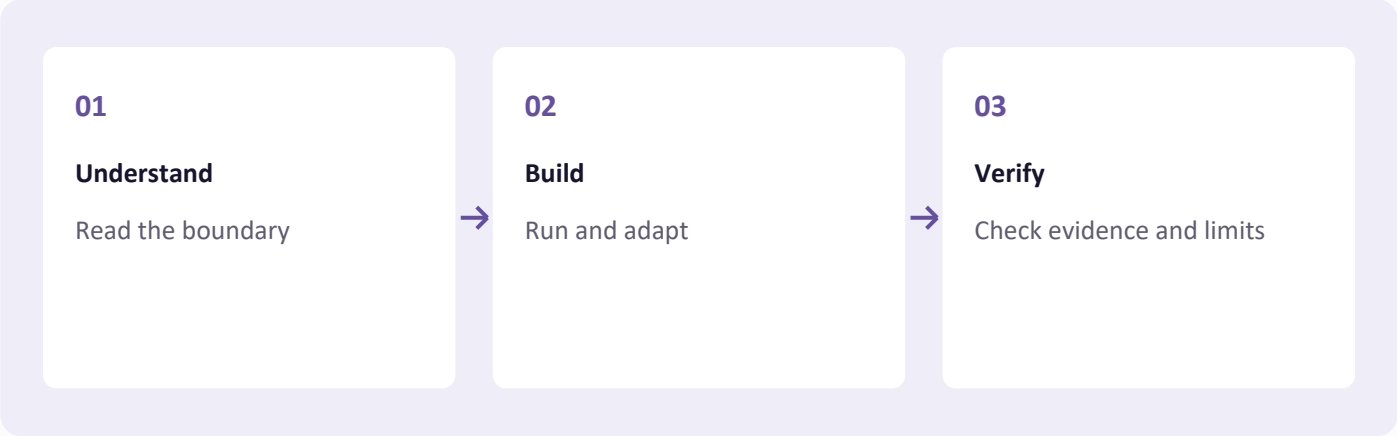
Workflow: an explicit sequence or graph of allowed steps. Fixed workflows are often easier to test than open-ended planning.

RAG: retrieval plus generation conditioned on selected evidence. Studio uses an extractive offline response stage as a transparent integration baseline.

Tool: an allowlisted application operation with a validated contract. A tool name is not a permission grant.

Evaluation: a defined case set, rubric, and measurement process. A passing unit test is not a general quality or safety guarantee.

Approval: informed permission for an exact proposed action, checked again at commit. A hash binds payload identity but does not authenticate a person.



01 / Map the complete request path

01 / UNDERSTAND THE BOUNDARY

A generative application is a chain of ordinary software boundaries around a model. The browser captures intent; an API authenticates and validates; retrieval selects context; a provider returns text or proposed actions; application code decides what may run. The model is one component, not the owner of the application. Start with a visible request path before choosing a framework. In this book the product is a small knowledge assistant called Studio: it answers questions about a fictional handbook and can propose a review note.



READ THE MODEL

Follow the arrows as data movement. Identify which component owns the rule and which component supplies untrusted input. Capability and permission are separate from the shape of the data.

Choose the contract

Separate responsibilities by the evidence each component can inspect. A browser cannot protect a provider key. A model cannot authenticate the signed-in user merely by reading a name in a prompt. A database cannot decide whether a generated answer is faithful. Give each boundary a small input and output contract. The demonstration traces stages without calling a model; that makes missing steps visible before any network service is introduced.

PUBLIC ENTRY POINTS IN THE REFERENCE

```
def request_path(question)
```

BROWSER

Intent and feedback. Locate the corresponding responsibility in the complete source on the reference page. Mark whether it is enforced by code or remains a design assumption.

PYTHON API

Validate and orchestrate. Locate the corresponding responsibility in the complete source on the reference page. Mark whether it is enforced by code or remains a design assumption.

MODEL AND DATA

Return evidence. Locate the corresponding responsibility in the complete source on the reference page. Mark whether it is enforced by code or remains a design assumption.

Demonstration and observed output

Run code/examples/01_map_the_complete_request_path.py

DRIVER / DEFINITIONS ARE IN THE MATCHING FILE

```
print(request_path("  How do reviews work?  "))
```

OBSERVED STDOUT / LOCAL RELEASE CHECK

```
{'question': 'How do reviews work?', 'stages': ['validate', 'retrieve',
'generate', 'review']}
```

TRACE THE CALL

Read the driver from top to bottom. Identify the supplied fixture, the function call, and the printed return value. Compare the result with the contract before changing an input. This stdout is observed from the distributed program, not invented model output.

PREDICT ONE CHANGED INPUT BEFORE RUNNING IT

Build the lab

Draw the path from submit button to answer. Add one failure outcome at every hop. Implement a request summary that normalizes the question and emits the four stage names in order. Keep the original user text out of operational logs. Accept only a nonempty string of at most 2,000 characters after trimming. Test the blank input, wrong type, and exactly-at-limit input before connecting a provider.



INPUTS, TYPES, UNITS, AND BOUNDS

OUTPUTS AND REJECTION POLICY

ALGORITHM AND FIRST FAILURE TEST

Reference implementation

code/solutions/01_map_the_complete_request_path.py

```
def request_path(question):
    if not isinstance(question, str):
        raise ValueError("question must be text")
    text = question.strip()
    if not 1 <= len(text) <= 2000:
        raise ValueError("question length")
    return {"question": text, "stages": [
        "validate", "retrieve", "generate", "review"
    ]}
```

COMPARE BEHAVIOR BEFORE STYLE

Run the full fixtures before refactoring. A different implementation is acceptable when it preserves the stated contract. Keep design assumptions and rejected behavior explicit; do not broaden capabilities merely to make an input pass.



Check the behavior

These three trusted verification blocks ran against the matching solution. Setup statements and negative cases are included. The runner supplies raises(expected_exception, function, *arguments). Run without -O so Python assertions stay enabled.

BLOCK 1 / PASSED IN THE RELEASE FIXTURE

```
assert request_path(" x ")["question"] == "x"
```

BLOCK 2 / PASSED IN THE RELEASE FIXTURE

```
raises(ValueError, request_path, " ")
```

BLOCK 3 / PASSED IN THE RELEASE FIXTURE

```
raises(ValueError, request_path, 5)
```

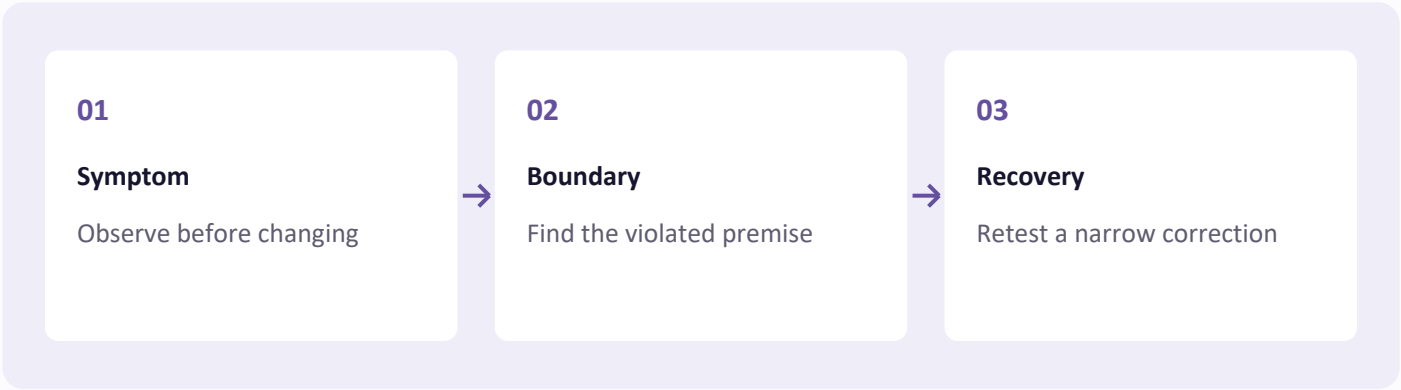
INTERPRET THE RESULT

A passing check supports the specific input and assertion shown. It does not establish all possible inputs, live model quality, deployment security, or a performance guarantee. Add a test for the first behavior your own application changes.

ONE MISSING CASE YOU WOULD ADD

Failure case and recovery

An answer can look fluent even when retrieval was skipped. A green HTTP status proves delivery, not grounding. Inspect the trace stages and the selected source identifiers before assessing prose. If the same component performs validation, retrieval, and rendering, a later change may silently bypass one of them. The small trace is an architectural sketch, not a distributed tracing implementation.



OBSERVED SYMPTOM AND SMALLEST REPRODUCTION

CAUSE: CODE DEFECT OR DOCUMENTED LIMIT?

CORRECTION, REGRESSION CHECK, AND REMAINING RISK

Transfer the idea

Replace the four strings with typed events when the app grows. Record durations and error categories, not raw questions by default. For a support assistant add escalation after review; for a drafting assistant add a preview and approval boundary before publishing. The request path should describe what your code actually enforces, not an aspirational diagram.

CONNECT IT TO THE SYSTEM

Identify where this behavior belongs in your browser, API, provider, retrieval, tool, or storage layer. Keep application policy outside untrusted generated content. State one assumption that must change before using the technique with real users.

PRIMARY REFERENCE / P1

Python tutorial and data structures

https://docs.python.org/3.12/tutorial/

YOUR ADAPTATION AND ITS FIRST ACCEPTANCE CHECK