11

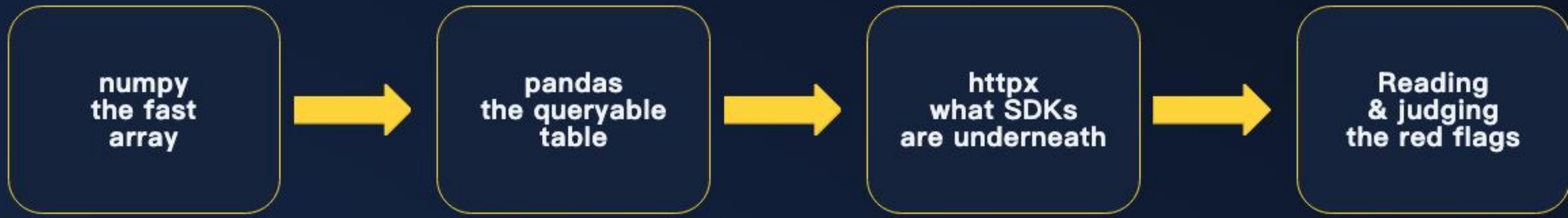
The Data & HTTP Layer

numpy, pandas, httpx — the code every AI pipeline is soaked in

Chapter 11 · you will READ far more of this than you write — so learn to judge it

11 The journey — three libraries, one job: read them

You meet these three in every RAG pipeline, every eval harness, every embeddings notebook. The goal is not to write them fluently — it is to understand what they do to your vectors and data, and spot when they do it badly.



By the end you can read an embeddings, eval, or API-call file and know what is really happening — and what a careful reviewer would flag.

11

numpy — the array that exists because lists are slow

Why embeddings and tensors are never plain lists

A Python list

A box of pointers to arbitrary objects, scattered in memory. Math on a million floats = a million separate Python operations. Slow, and heavy on memory.

A numpy array (ndarray)

One flat block of raw numbers, all the same type, packed side by side in memory, with the math done in C. That is why it is fast at scale.

For embeddings, tensors and similarity scores the speed is not optional — that is why they are all numpy arrays, never lists.

11

shape and dtype — the two things you always check

A wrong shape is the number-one bug in AI glue code

```
import numpy as np

v = np.array([0.1, 0.2, 0.3, 0.4])
v.shape    # => (4,)      one dimension, 4 elements
v.dtype    # => float64   every element is the same type

m = np.array([[1.0, 0.0, 0.0, 0.0],   # 3 embeddings,
              [0.0, 1.0, 0.0, 0.0],   # each of length 4
              [0.5, 0.5, 0.0, 0.0]])
m.shape    # => (3, 4)    3 rows, 4 columns
```

shape

The size along each dimension. (4,) is a flat vector; (3, 4) is 3 rows by 4 columns.

dtype

The element type (here float64). One fixed type for the whole array — that is the trade for speed.

A silent shape mismatch that happens to line up produces WRONG NUMBERS, not an error — so shapes get checked at model boundaries.

11

Vectorized operations — the whole point

Apply the maths to the ENTIRE array at once, no loop

```
a = np.array([1.0, 2.0, 3.0])
b = np.array([10.0, 20.0, 30.0])

a + b      # => [11. 22. 33.]   element-wise, no loop
a * 2      # => [2. 4. 6.]      the 2 applied to every element
a @ b      # => 140.0           dot product (1*10 + 2*20 + 3*30)
```

numpy runs the loop in C. The pure-Python version [x + y for x, y in zip(a, b)] gives the same answer but is often 10 to 100 times slower on real-sized arrays.

Red flag when reading

A Python for loop doing arithmetic element-by-element over arrays throws away numpy's entire reason for existing. It is a performance bug.

11

Broadcasting and slicing — recognise, don't author

Two more idioms you will see on every line

Broadcasting

The rule that lets arrays of DIFFERENT shapes combine — numpy stretches the smaller one to fit, without copying. a * 2 already did it: the scalar 2 was spread across every element. A (4,) row added to a (3, 4) matrix applies to all 3 rows.

```
m[0]        # => [1. 0. 0. 0.]  first row
m[0, 2]     # => 0.0            row 0, column
2
m[:, 1]     # => [0. 1. 0.5]    a whole
column
m[0:2]      # => first two rows
```

Slicing looks like a list but a comma addresses more than one dimension. m[:, 1] means 'every row, column 1' — there is no JavaScript equivalent; you would write a .map().

11

Cosine similarity — the workhorse of search

When a system 'finds the most relevant chunk', this runs

```
def cosine_similarity(a, b):
    # dot product / product of the vectors' lengths (norms)
    return float(a @ b / (np.linalg.norm(a) * np.linalg.norm(b)))

query = np.array([0.9, 0.1, 0.0])
doc   = np.array([0.8, 0.2, 0.0])
cosine_similarity(query, doc)   # => 0.98...  very similar
```

It measures the ANGLE between two embedding vectors. 1.0 = identical direction, 0.0 = unrelated. Two short lines of numpy.

For many documents at once, the batched form docs @ query does it all in one matrix multiply — same idea, no loop.

11

pandas — the spreadsheet as a variable

Where you meet it most: evaluation datasets

DataFrame = a table

Labelled rows, named columns, a different type allowed per column — that you manipulate in code. A single column is called a Series.

```
import pandas as pd

df = pd.read_csv('eval_results.csv')
df.head()   # peek at the first rows — ALWAYS do this first

#  id   prompt            model    latency_ms  score
#  1    Summarize this... gpt-5.1          820  0.91
#  2    Translate to fr.. gpt-5.1          640  0.88
```

If numpy is the array, pandas is the spreadsheet you can query. In GenAI work it holds your test prompts, expected answers and model scores — the table that tells you if your system actually works.

11

Selecting rows, and the boolean mask

The filter pattern you must be able to read

```
df['score']                # one column (a Series)
df[['prompt', 'score']]    # two columns (note the double brackets)
df.loc[0, 'prompt']        # by LABEL: row 0, column 'prompt'
df.iloc[0]                 # by POSITION: the whole first row
```

```
# df['score'] < 0.5 is a Series of True/False.
# Pass it back into df[...] to keep only the True rows:
failures = df[df['score'] < 0.5]

# Combine with & (and) / | (or) — parenthesise EACH
condition
df[(df['score'] < 0.5) & (df['latency_ms'] > 1000)]
```

A boolean mask

A column of True/False used to pick rows. df[df.score < 0.5] keeps the failures worth inspecting — like records.filter in JS, but it runs in C over the whole column.

11

groupby and apply — the heart of an eval report

Turn a table of results into a summary

```
df['score'].mean()        # => 0.87   average across all rows
df['latency_ms'].max()    # => 2100

# Average score PER MODEL — the core of any comparison
df.groupby('model')['score'].mean()
# gpt-5.1             0.87
# claude-sonnet-4-5   0.89
```

groupby('model') splits the table into one group per model; ['score'].mean() averages within each. This three-token line is most of what an eval report is.

```
# .apply runs a function on each row/value when there is no
# built-in vectorized form for it:
df['grade'] = df['score'].apply(lambda s: 'pass' if s >= 0.8 else 'fail')
```

11 pandas — judging it well

There is no first-class pandas in JavaScript

JS -> Python

Closest mental model: an array of objects you query — except pandas works COLUMN-first and vectorized. df[df.score < 0.5] is like records.filter(r => r.score < 0.5), but in C over whole columns. Think 'a SQL result set', not 'a plain array'.

Red flags when reading

1. A whole giant CSV read into memory — pd.read_csv('huge.csv') for multi-GB logs needs chunking or a database.

2. Long chained ops with no named steps — correct but opaque; a clear pipeline names each stage.

3. pandas in a per-request web handler — it is for wrangling, not production request paths.

load -> filter -> transform -> group -> summarize. Reading a data-prep pipeline is just following the DataFrame through that chain; each step returns a new one.

11

HTTP with httpx — what every SDK is underneath

Every LLM call is an HTTPS POST with a JSON body

```
import httpx

resp = httpx.get('https://api.example.com/models',
timeout=10.0)
resp.status_code    # => 200
resp.json()         # parse the JSON body into a Python
dict/list

resp = httpx.post(
    'https://api.example.com/v1/chat',
    headers={'Authorization': 'Bearer sk-...'},
    json={'model': 'gpt-5.1', 'messages': [...]},
    timeout=30.0)
```

The SDKs hide it — but when you debug a timeout, a 429 rate limit or a hanging stream, you are back here at the HTTP layer.

requests is the classic sync library. httpx is the modern one — same thing plus async and HTTP/2 — and it is what the OpenAI and Anthropic SDKs use inside.

11 raise_for_status and timeout — two lines that matter most

Skip either and AI code fails in baffling ways

raise_for_status()

Turns a failed status (401 auth, 429 rate limit, 500 server) into a Python exception. Without it, resp.json() happily parses an {'error': ...} body as if it were success — producing baffling downstream failures.

timeout

Bounds how long you will wait. Without it, a stalled connection hangs your program FOREVER — which in an agent loop means a wedged worker that never recovers. Every request to an external service must set one.

```
resp.raise_for_status()    # raise on any 4xx / 5xx status
data = resp.json()
```

11

Reusing connections, and streaming

A pooled client, and how token-by-token output arrives

```
# Reuse ONE connection pool across many requests
with httpx.Client(timeout=30.0, headers={...}) as client:
    for prompt in prompts:
        r = client.post(url, json={'input': prompt})
        r.raise_for_status()

# Async version — for firing many calls concurrently
async with httpx.AsyncClient() as client:
    r = await client.post(url, json={'input': 'Hi'})
```

A fresh connection per request is wasteful. A Client (sync) or AsyncClient (async) pools and reuses connections; the with block guarantees it is closed.

Streaming

client.stream(...) + for line in resp.iter_lines() processes each chunk as it arrives, without buffering the whole body. The SDK's 'for chunk in stream' is exactly this, dressed up.

11 HTTP red flags, and the whole-chapter teaching

Missing timeout — one call with no timeout can hang an entire agent.

Missing raise_for_status — reads an error body as if it were success.

Unbounded concurrency — hundreds of AsyncClient calls at once, no semaphore.

A new Client() built inside a hot loop instead of reused once.

An auth token hard-coded in the source instead of read from the environment.

How to read this whole layer

numpy: watch loops doing array maths, and shapes.

pandas: watch whole-file reads and opaque chains.

httpx: watch the missing timeout and the unchecked status.

Each smell is short to spot and expensive to miss.

The teaching — you will read far more numpy, pandas and httpx than you write. Learn what each does and its handful of red flags, and you can judge the code that every RAG pipeline, eval harness and LLM SDK is built from.