09

Iterators, Generators & Async

How LLM streaming and concurrency actually work

Chapter 9 · the two threads that run through every agent codebase

09 The journey — two clusters, five ideas

You meet both on day one of reading real LLM code. Lazy evaluation explains streaming tokens; async explains firing 50 calls at once. Learn both and the magic disappears.

Stops 1–3 are lazy evaluation (streaming); stops 4–5 are async (fanning out calls). Same theme both times: do work only when it is needed.

09

Iterables vs iterators

Python draws a line JavaScript blurs

Iterable

Anything you CAN loop over — a list, a string, a file. It knows how to hand out a cursor.

Iterator

The one-shot cursor that actually walks through it, remembering where it is. Used up once done.

```
nums = [10, 20, 30]        # a list: iterable, but NOT an iterator
it = iter(nums)            # iter() asks for a fresh cursor
next(it)                   # => 10
next(it)                   # => 20
next(it)                   # => 30
next(it, 'DONE')           # => DONE   (2nd arg = default, no error)
```

iter() gives you the cursor; next() pulls one value at a time. A list hands out a NEW cursor each time you ask.

09

A 'for' loop is just iter + next

Two dunder methods define the whole protocol

```
__iter__(self)  returns an iterator (a fresh cursor).
```

```
__next__(self)  returns the next value, or raises StopIteration.
```

```
for n in nums:      # this loop...
    print(n)

# ...is EXACTLY: call iter(nums) once, then next() over and over,
# stopping silently when StopIteration fires. That is the whole thing.
```

JS to Python: it's JS's iteration protocol with renamed parts. Symbol.iterator becomes __iter__; .next() returning {value, done} becomes __next__ returning a value OR raising StopIteration. JS flags done: true; Python raises. StopIteration is a signal, not an error.

09

Generators — the easy way with 'yield'

Any def containing yield builds the iterator for you

Calling a generator runs NO body code — it hands back a generator object. Each next() runs until the next yield, PAUSES there (freezing all local state), and resumes on the following next().

```
def countdown(n: int):
    while n > 0:
        yield n          # pause here, hand out n, remember
everything
        n -= 1

gen = countdown(3)
next(gen)           # => 3
list(gen)           # => [2, 1]    (drains the rest)
```

The payoff:
laziness + memory

A generator holds ONE item at a time, so you can process data far larger than RAM.

09

The pattern you see everywhere

Stream pages from an API — memory stays flat

```
def fetch_all_documents(client, batch_size=100):
    cursor = 0
    while True:
        page = client.get_documents(offset=cursor, limit=batch_size)
        if not page:            # empty page = we're done
            return              # bare return ends a generator
        for doc in page:
            yield doc           # caller sees a flat stream of docs
        cursor += len(page)
```

You loop as if it is one endless list — but only about 100 rows ever live in memory at once. The caller writes for doc in fetch_all_documents(client): and never sees the paging.

09

Gen expressions, yield from, infinite

Three more shapes you'll recognise

```
[x*x for x in range(1_000_000)]   # list: ~8 MB allocated NOW
(x*x for x in range(1_000_000))   # gen expr: near-zero, lazy
sum(x*x for x in range(5))        # => 30   (no [] needed in a call)
```

```
def chain_streams(*streams):
    for s in streams:
        yield from s      # re-yield every item of s
(flatten)
list(chain_streams([1, 2], [3, 4]))   # => [1, 2, 3, 4]
```

A generator expression = a comprehension with ( ) instead of [ ]. It builds NOTHING until you iterate it.

Generators can be INFINITE

```
def ids():
    n = 0
    while True:
        yield n
        n += 1
```

Legal precisely because they're lazy — safe only if the consumer stops pulling.

09

The key idea — streaming IS an iterator

Why streamed replies feel live

When you write for chunk in stream: against a chat completion, the SDK is a generator handing you deltas as bytes arrive over the wire. Each yield is one token-ish chunk. You never hold the full response — you print pieces as they land.

```
stream = client.chat.completions.create(
    model='...', messages=[...], stream=True)
for chunk in stream:                    # <-- iterator protocol, nothing new
    delta = chunk.choices[0].delta.content
    if delta:
        print(delta, end='', flush=True)    # print tokens as they arrive
```

Nothing new to learn — it is exactly the iter / next mechanism from the last few slides.

09

itertools + a laziness trap to spot

Three lazy tools to recognise

islice(it, n)
take the first n items of any iterator — safely sip an infinite one.

chain(a, b, ...)
concatenate iterables lazily (the function form of yield from).

tee(it, k)
split one iterator into k independent ones — but it BUFFERS what the slowest branch hasn't read.

Reading & judging

list(some_generator) at the top of a function quietly CANCELS the laziness — the whole stream is now in memory. Fine for 100 rows, a bug for 10 million.

See a generator wrapped in list() or sorted() right before a for? Ask whether streaming was the point. And tee on an unbounded stream can grow memory without limit.

09

Async — the why: waiting on I/O

An LLM call spends ~2 seconds doing nothing locally

It sent bytes and is WAITING for the model. HTTP requests, database queries, file reads are all like this — I/O-bound. Blocking one at a time wastes the wait.

Threads help I/O but Python's GIL (Global Interpreter Lock) lets only ONE thread run Python at a time — plus locking headaches.

Async's route: cooperative concurrency on ONE thread. One event loop runs your coroutines; when a coroutine hits await on I/O it VOLUNTARILY steps aside so the loop runs another, resuming the first when its data is ready. Async is NOT parallelism — one worker juggling many waits.

JS to Python: you already own this. Python's event loop IS Node's event loop. A coroutine (from async def) is a Promise/awaitable. await is await. asyncio.run(main()) is the top-level bootstrap Node does for you.

09

async def, await, and the top bug

Calling a coroutine does NOT run it

```
async def get_answer(prompt: str) -> str:
    await asyncio.sleep(1)     # pretend this is an LLM call
    return f'answer to: {prompt}'

maybe = get_answer('hi')       # THE classic mistake
print(maybe)   # => <coroutine object ...>   NOT the result!
```

```
async def main():
    result = await get_answer('hi')   # now it actually runs
    print(result)                     # => answer to: hi

asyncio.run(main())   # the ONE sync entry point that starts
the loop
```

JS to Python: sharper edge.

In JS, calling an async function STARTS it. In Python the coroutine is COMPLETELY INERT until awaited — forget the await and nothing happens at all, silently.

The single most common async bug in GenAI code.

09

Running things concurrently

Schedule first, then await together

```
async def sequential():
    for p in ['a', 'b', 'c']:
        out.append(await call_llm(p))   # wait fully each
time
    # => 3.0s   (3 x 1s, one after another)

async def concurrent():
    out = await asyncio.gather(          # fire all three,
overlap
        call_llm('a'), call_llm('b'), call_llm('c'))
    # => 1.0s   (the waits overlap!)
```

gather(*coros)

Runs them concurrently, returns results in ARGUMENT order (not completion order).

Fanning out many independent model calls is THE biggest real-world async win — a 3x to Nx speedup.

JS to Python: asyncio.gather(a, b, c) is Promise.all([a, b, c]) — same ordered-results guarantee.
asyncio.create_task(coro) is eagerly starting a Promise in the background.

09

TaskGroup & timeout — the modern way

Structured concurrency (Python 3.11+)

```
async def fan_out(prompts: list[str]) -> list[str]:
    async with asyncio.TaskGroup() as tg:        # 3.11+
        tasks = [tg.create_task(call_llm(p)) for p in prompts]
    # on exit, all tasks are done (or the group raised)
    return [t.result() for t in tasks]
```

```
async def guarded():
    try:
        async with asyncio.timeout(0.5):     # budget 0.5s
            return await call_llm('slow')    # needs ~1s
    except TimeoutError:
        return 'timed out'
```

asyncio.sleep(n) is the async-friendly pause — it yields to the loop. time.sleep freezes everything.

TaskGroup beats bare gather: if one task raises, the group CANCELS its siblings and propagates the error cleanly.

Add a timeout so a hung API call cannot stall forever.

09

Async iteration & streaming

The async cousins of Part A

async for — iterate an async iterator; each step may await.
async with — a context manager that can await (open/close an HTTP connection).
async generator — a coroutine with yield: it produces values AND can await between them.

```
async def token_stream(text: str):
    for word in text.split():
        await asyncio.sleep(0.1)    # network delay per token
        yield word                  # async generator: await +
yield

async def consume():
    async for tok in token_stream('streaming is async
iteration'):
        print(tok, end=' ', flush=True)
```

This is exactly the real-SDK shape:

async for chunk in stream:

Combine both wins — stream each response, but fan out N prompts with gather. Best of both.

JS: async for...in is JS for await...of; an async generator is JS async function*.

09

Sync vs async clients — how to tell

The keywords are the only reliable signal

Sync (blocking)		Async (awaitable)
`OpenAI()`	->	`AsyncOpenAI()`
`Anthropic()`	->	`AsyncAnthropic()`
`httpx.Client()`	->	`httpx.AsyncClient()`
`client.create(...)`	->	`await client.create(...)`
`for chunk in stream:`	->	`async for chunk in stream:`

The tell

An Async... class name, an await before the call, or async for / async with = async code that lives inside asyncio.run.

No await and a plain for chunk in stream: = sync code that BLOCKS the thread.

The API is mirror-imaged on purpose — trust the async / await keywords, not the method names.

Pitfall: the loop is one cooperative thread, so anything that does not await HOGS it. time.sleep, sync requests / httpx, or a big CPU loop inside async def freezes EVERY coroutine. Fix: async equivalents, or await asyncio.to_thread(fn) to offload blocking work.

09 Red flags & the close

Five async red flags to hunt:

1. Un-awaited coroutine — a create(...) with no await. Silent no-op; the call never happens.

2. Blocking the loop — time.sleep, sync HTTP, or heavy CPU inside async def. One offender freezes all.

3. Sequential awaits that should be gathered — a 50-item await loop is 50x too slow.

4. Materializing where streaming was the point.

5. Unbounded concurrency — gather over 10,000 prompts with no Semaphore. Hits rate limits (429s).

```
# The rate-limit-safe fan-out you WANT
to see:
async def bounded_call(sem, prompt):
    async with sem:             # at
most N in flight
        return await call_llm(prompt)

async def safe_batch(prompts):
    sem = asyncio.Semaphore(5) # cap at
5
    return await asyncio.gather(
        *(bounded_call(sem, p) for p in
prompts))
```

The teaching — lazy iterators stream the tokens; async concurrency fans out the calls. Hold these two threads and virtually every LLM and agent codebase you read stops looking like magic.