Advanced asyncio, Practically

Reliable concurrency with modern Python

SmNix

2026

Contents

Introduction

`asyncio` looks simple in a tutorial: mark a function `async`, add a few `await` expressions, and run it on an event loop. Production systems expose a different problem. Work overlaps, one dependency slows down, queues fill, clients disconnect, deployments interrupt requests, and cleanup must still finish.

This book is about those moments.

It targets Python 3.11+ and assumes you can read basic coroutines and have used `asyncio.run()`. You don't need experience building an async service. Each chapter opens with a concrete problem, develops a mental model, examines a failure mode, and ends with practices you can apply immediately.

One promise runs throughout: every claim here is either demonstrated by a program in `examples/` that you can run, or it is marked as a judgement call. Where common practice and I disagree about what is idiomatic, I say so and explain why.

What you will build

The running case study is **Price Watcher**, a small service that:

- accepts product identifiers,
- queries several suppliers concurrently,
- applies per-request deadlines,
- limits supplier concurrency and queue growth,
- isolates a blocking legacy client,
- records enough context to debug failures,
- keeps one bad job from destroying a batch, and
- drains accepted work during shutdown.

The examples isolate each technique so you can run and modify it without installing a framework. Chapter 8 assembles them into one service design.

How to use this book

Read Chapters 1–3 in order. Task ownership and cancellation underpin everything that follows, and the later chapters assume both. Chapters 4–7 are independent of each other. For a fast production review, jump to Chapter 10 and run its checklist against your own application.

The examples use only the standard library. Run them as you read:

```
python examples/structured_concurrency.py
python examples/deadlines.py
python examples/backpressure.py
python examples/blocking_work.py
python examples/graceful_service.py
python examples/observability.py
python examples/price_watcher.py
python examples/diagnostics.py
```

Then run the tests, which are themselves worked examples of testing concurrent code:

```
python -m unittest discover -s tests -v
```

On Windows use `python`; on systems where Python 2 still owns that name, use `python3`. Every example targets 3.11 and is verified on 3.14.

Conventions

- **Invariant** marks a condition the design must preserve.
- **Failure mode** shows how reasonable-looking code breaks under pressure.
- **Decision** compares alternatives instead of declaring one universal tool.
- **Practical rule** is a habit worth adopting without further deliberation.
- **Try it** is an experiment to run before continuing.
- *Fragment* labels code that emphasises one idea and won't run alone. Every other listing is extracted from a tested program, and each chapter links to the file it came from.
- Durations in examples are deliberately short. Production values belong in configuration and should follow a measured service-level objective.

CHAPTER 1

Think in lifetimes

An event loop runs one Python task at a time. A task keeps running until it finishes or reaches an operation that actually suspends. `await` is therefore a result boundary and a *possible* scheduling boundary. An awaited coroutine may complete immediately without ever giving another task a turn.

Fragment:

```
async def misleading() -> None:
    expensive_pure_python_work()  # No other task can run here.
    await asyncio.sleep(0)        # Scheduling resumes only here.
```

Concurrency is useful when tasks spend much of their lives waiting for I/O. It does not make ordinary Python computation parallel.

One thread, many timelines

The default event loop uses cooperative scheduling. A task voluntarily gives control back when an operation cannot complete now: a socket has no bytes, a queue is empty, a timer has not expired, or a lock is held. The loop records what each task is waiting for, asks the operating system which resources are ready, and resumes the corresponding tasks.

Consider two coroutines:

Fragment:

```
async def fetch_price(supplier: str) -> int:
    print(f"start {supplier}")
    await network_read(supplier)
    print(f"finish {supplier}")
    return 100
```

If two instances are tasks, execution can interleave like this:

```
task A: print start A -> wait for socket A
task B: print start B -> wait for socket B
event loop: socket B becomes ready
task B: print finish B -> complete
```

```
event loop: socket A becomes ready
task A: print finish A -> complete
```

Nothing implies A finishes first because it started first, and no two Python statements here run at the same instant. The overlap happens during waiting.

That is the central tradeoff:

- switching tasks is cheap, because it doesn't require an operating-system thread per operation;
- fairness is cooperative, so one task that computes or blocks for too long delays every other task on that loop.

Invariant: code running on the event-loop thread must reach suspension points frequently enough to meet the application's latency target.

`await` does not create concurrency

This code is asynchronous but sequential:

Fragment:

```
profile = await fetch_profile(user_id)
orders = await fetch_orders(user_id)
```

The second call doesn't begin until the first finishes. If each takes 200 ms, the pair takes about 400 ms. To overlap independent work, schedule both operations before waiting for either result:

Fragment:

```
profile_task = asyncio.create_task(fetch_profile(user_id))
orders_task = asyncio.create_task(fetch_orders(user_id))

profile = await profile_task
orders = await orders_task
```

This pulls ideal elapsed time down toward the slower call, about 200 ms. It also introduces an ownership problem: what happens to `orders_task` if awaiting `profile_task` raises? Nothing good. The exception propagates, the function returns, and `orders_task` keeps running with nobody left to observe it. Chapter 2 replaces this hand-managed pattern with a structured scope, and you should prefer that scope in real code. The manual version appears here only because you will meet it in existing codebases.

Coroutine, task, and future

- A **coroutine object** describes work. Calling an `async def` function does not begin that work.
- A **task** schedules one coroutine on an event loop and owns its eventual result or exception.
- A **future** is a lower-level placeholder for a result. Application code usually consumes futures but rarely creates them.

Fragment:

```
coroutine = fetch_user(42)              # Not running yet
task = asyncio.create_task(coroutine)  # Scheduled now
user = await task                       # Observe its outcome
```

Every task should have an owner that decides when it starts, how long it may run, and how its result is observed. An unowned task is a source of leaked work and lost exceptions.

Task state and result observation

A task moves through a small lifecycle:

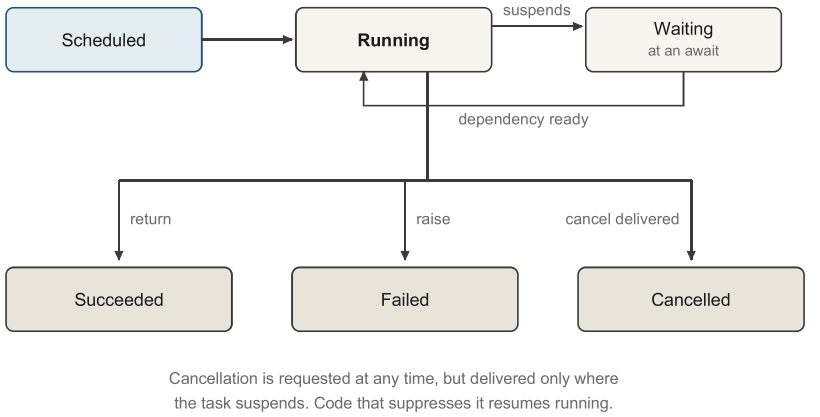

Figure 1.1 The task lifecycle. Every task ends in exactly one of three terminal states, and a cancellation request reaches it only where it suspends.

`task.done()` tells you whether the task reached a terminal state. Once done, `task.result()` returns the value or re-raises its exception;