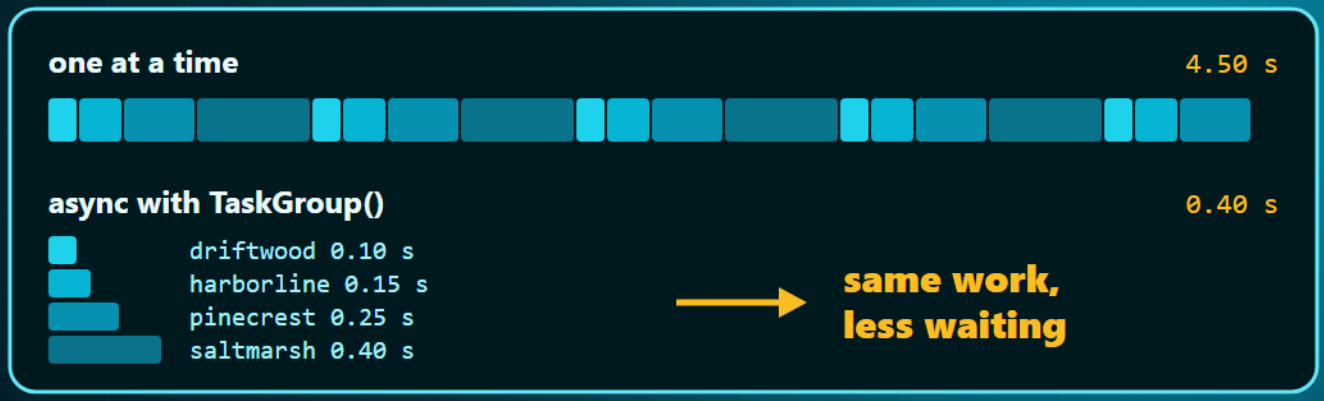



```python
async with asyncio.TaskGroup() as tg:
  for store in DEFAULT_STORES:
    tg.create_task(
      fetch_offer(client, store,
      "GW-103"))
async with asyncio.timeout(1.0): ...
```

Thread · Process · Task · Queue · Lock

PRICE ALERT

GW-103 · Headphones

Target	$185.00
Driftwood	$199.00
Harborline	$189.50
Pinecrest	**$184.99**

3 alerts · 18 offers · 4 stores

A COMPLETE, PRACTICAL GUIDE

Concurrency and Async in Python

Threads, processes, asyncio, and structured concurrency for faster, responsive programs

The GIL · threading · concurrent.futures · multiprocessing
TaskGroup · timeouts · httpx · aiohttp · pytest-asyncio

MEASURE · CHOOSE · BUILD · TEST · SHIP

THENEXBOOKS

Concurrency and Async in Python

Threads, Processes, asyncio, and Structured Concurrency for Faster, Responsive Programs From your first thread to a tested, production-ready concurrent application — built step by step

First Edition

THENEXBOOKS

Concurrency and Async in Python
Threads, Processes, asyncio, and Structured Concurrency for Faster, Responsive Programs

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. Python and its libraries change over time; behavior, defaults, and feature availability can differ between versions, builds, and operating systems. Timings shown in this book were measured on one machine and are illustrative only — your numbers will differ. Always consult the official documentation for the versions you use.

Python is a registered trademark of the Python Software Foundation. Windows is a trademark of the Microsoft group of companies. macOS is a trademark of Apple Inc. Linux is a registered trademark of Linus Torvalds. Other product and project names, including httpx, aiohttp, pytest, NumPy, and numba, are the property of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by the Python Software Foundation or any project mentioned.

Gullwing, the price-tracking application used as the running example in this book, is fictional, as are its team, the stores Driftwood Supply, Harborline Outfitters, Pinecrest Market, Saltmarsh Goods, and Fogbank Electronics, their products, prices, and all sample data. Every network request in the examples goes to a local fake server on your own computer.

Preface

Most programs spend most of their time waiting. They wait for a web server to answer, for a database to return rows, for a file to arrive, for a user to click. While one request is in flight, the processor sits idle, and a program that asks twenty slow servers one at a time takes twenty times as long as it needs to. Other programs have the opposite problem: they crunch numbers on one core while the rest of the machine does nothing.

Python gives you three serious tools for these problems — threads, processes, and `asyncio` — and each one comes with its own rules, traps, and folklore. Threads are easy to start and easy to get wrong. Processes sidestep the global interpreter lock but charge you for every byte they exchange. `asyncio` can juggle thousands of connections on a single thread, but one blocking call freezes everything, and cancellation has semantics you must understand before you trust it in production.

This book teaches all three from the ground up and, just as importantly, teaches you how to choose between them. We build one real application step by step, measure every claim we make on a real machine, show each failure — the race condition, the frozen event loop, the orphaned task, the deadlock — before we fix it, and finish with code that is structured, tested, and ready to run unattended.

Who This Book Is For

- **Intermediate Python developers** who are comfortable with functions, classes, modules, virtual environments, and basic pytest, and want to make their programs faster and more responsive.
- **Backend and automation engineers** who call many services, scrape or poll APIs, process queues, or run scheduled jobs and need them to finish in seconds instead of minutes.
- **Data and tooling developers** who want to use every CPU core for heavy computation without fighting the GIL.
- **Developers who have been burned** by flaky threaded code or mysterious `asyncio` errors and want a clear mental model instead of more trial and error.

What You Will Learn

- The difference between concurrency and parallelism, how to tell I/O-bound from CPU-bound work, and how to measure before you optimize.
- What the GIL really does, when it is released, and what free-threaded builds of recent Python versions change — and what they do not.
- Threads done right: starting and joining, race conditions, locks, queues, events, semaphores, and thread pools with `concurrent.futures`.

- Processes done right: the spawn start method, pools, pickling costs, shared memory, and hybrid designs.
- `asyncio` from first principles: coroutines, the event loop, tasks, `TaskGroup` and structured concurrency, exception groups, timeouts, and cancellation.
- Real async I/O with httpx and aiohttp: connection pools, semaphores, rate limiters, retries with backoff, circuit breakers, and producer/consumer pipelines.
- Mixing sync and async code safely with `to_thread` and executors, async patterns for databases and files, and a small async server of your own.
- Testing with pytest and pytest-asyncio, debugging with asyncio debug mode and `faulthandler`, profiling with pyinstrument, and running a concurrent service in production with graceful shutdown.

How This Book Is Organized

Part I — Foundations (Chapters 1–5) explains why concurrency matters, how to measure, how the GIL works, and how to write correct threaded code. **Part II — Threads and Processes** (Chapters 6–10) covers coordination primitives, `concurrent.futures`, multiprocessing, shared state, and how to choose between threads and processes. **Part III — asyncio Fundamentals** (Chapters 11–15) builds `asyncio` from coroutines up to tasks, `TaskGroup`, timeouts, cancellation, and async iteration. **Part IV — Async I/O in Practice** (Chapters 16–20) puts it to work with async HTTP clients, concurrency limits, resilience patterns, pipelines, and an async server. **Part V — Mixing Worlds** (Chapters 21–24) bridges sync and async code, handles databases and files, and covers testing and debugging. **Part VI — Performance and Production** (Chapters 25–28) covers profiling, choosing the right model, production patterns, and a complete capstone. The appendices provide a cheat sheet, templates and reference tables, a production checklist, and a glossary.

Each chapter ends with a **Try It Yourself** section and a **Key Takeaways** box for quick review.

The Running Example: Gullwing

Throughout the book we build **Gullwing**, a fictional price tracker. Gullwing watches five products — from a hiking backpack to a cast iron skillet — across four online stores, finds the best in-stock price for each, and sends an alert when a price drops to the target a user has set. The stores are not real: they are served by a small fake store server that runs on your own computer, with deliberately slow responses, missing products, out-of-stock items, a flaky store that fails on the first try, a strict mode that rejects impolite clients, and one very slow store for timeout experiments. That makes every experiment repeatable: polling the four stores one request at a time takes about 4.5 seconds by design, and you will watch that number fall as each technique is applied. Along the way Gullwing grows a thread-safe price board, a CPU-heavy analysis step, rate limiting, retries, a circuit breaker, a queue-based pipeline, SQLite snapshots, a test suite, and a polling service that shuts down cleanly.

Versions and Conventions

Examples target **Python 3.12 and later**, and every example was run on Python 3.12. Features added in Python 3.13 and 3.14 — such as `Queue.shutdown()`, the async-iterator form of `as_completed`, free-threaded builds, and subinterpreters — are explained in Notes and clearly labeled with the version that introduced them. Third-party libraries used: httpx, aiohttp, requests, pytest, pytest-asyncio, hypothesis, NumPy, numba, AnyIO, and pyinstrument; recent versions of each should work. The examples run on Windows, macOS, and Linux; where an operating system behaves differently (process start methods, signal handling), the text says so.

Timings are real measurements from one machine and are shown as "about" values. Your numbers will differ, but the shape of the results — what gets faster, and why — will not. Money is stored as integer cents and printed in US dollars, such as `$84.99`; dates use the ISO format `2026-03-14`.

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

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

> **WARNING**
> A common mistake or something that can cause hangs, data corruption, or lost work.

Code appears in `monospace` with a caption naming the file, such as `gullwing/fetch.py`; commands are captioned **Terminal** and program output is captioned **Output**. Commands use `python`; on Windows, use `py` if `python` is not on your PATH. Long lines are wrapped to fit the page.

Contents

Preface

PART I — FOUNDATIONS

1 Why Concurrency: Waiting, Working, and the Gullwing Price Tracker

2 Concurrency vs. Parallelism: I/O-Bound, CPU-Bound, and Measuring First

3 Inside CPython: The GIL, Free-Threaded Builds, and What They Mean for You

4 Threads with the threading Module: Starting, Joining, and Daemon Threads

5 Thread Safety: Race Conditions, Locks, and the Atomicity Myth

PART II — THREADS AND PROCESSES

6 Coordinating Threads: Queues, Events, Conditions, and Semaphores

7 concurrent.futures: Executors, Futures, and Thread Pools Done Right

8 Multiprocessing: Processes, Pools, and the Spawn Start Method

9 Sharing State Between Processes: Queues, Pipes, Shared Memory, and Managers

10 Threads or Processes? CPU Work, Pickling Costs, and Hybrid Designs

PART III — ASYNCIO FUNDAMENTALS

11 asyncio from First Principles: Coroutines, await, and the Event Loop

12 Tasks: Scheduling, gather, as_completed, and Task Lifetimes

13 Structured Concurrency with TaskGroup and Exception Groups

14 Timeouts and Cancellation: asyncio.timeout, CancelledError, and Cleanup

15 Async Iteration and Context: Async Generators, async for, and async with

PART IV — ASYNC I/O IN PRACTICE

16 Async HTTP with httpx and aiohttp: Clients, Sessions, and Connection Pools

17 Limiting Concurrency: Semaphores, Rate Limiters, and Polite Clients

18 Resilience: Retries, Backoff, Partial Failure, and Circuit Breakers

19 Producer/Consumer Pipelines with asyncio.Queue

20 Async Servers and Streams: Building the Fake Store with aiohttp

PART V — MIXING WORLDS

21 Bridging Sync and Async: to_thread, run_in_executor, and Calling Async from Sync

22 Async Data Access: Databases, Files, and Blocking Libraries

23 Testing Concurrent and Async Code with pytest and pytest-asyncio

24 Debugging Concurrency: Debug Mode, Deadlocks, Leaks, and Logging

PART VI — PERFORMANCE AND PRODUCTION

25 Profiling and Measuring Concurrent Programs

26 Choosing the Right Model: A Decision Guide and What Comes Next

27 Production Patterns: Configuration, Graceful Shutdown, and Observability

28 Capstone: Shipping Gullwing, a Concurrent Price Tracker

APPENDICES

A Concurrency and asyncio Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

Foundations

Why programs wait and how concurrency helps, the difference between concurrency and parallelism, measuring before optimizing, what the GIL really does and what free-threaded builds change, and writing threaded code that is actually correct.

1 · Why Concurrency

2 · Concurrency vs. Parallelism

3 · Inside CPython: The GIL

4 · Threads with the threading Module

5 · Thread Safety

CHAPTER 1 · PART I — FOUNDATIONS

Why Concurrency: Waiting, Working, and the Gullwing Price Tracker

Most slow programs are not busy; they are waiting. In this chapter we meet Gullwing, the price tracker we will build throughout the book, run it three different ways, and see the same work finish in about a third of the time without a faster computer, a faster network, or a single clever algorithm.

Programs Spend Most of Their Time Waiting

Think about what a typical Python program does in a day. It reads a file, calls a web API, queries a database, writes a log line, sends an email, and waits for a user to click something. Between those steps it does some computation: it parses JSON, compares numbers, builds a report. If you could watch the processor while that program runs, you would see something surprising. The processor is idle most of the time. It sent a request across the network and now it sits there, doing nothing useful, until the answer comes back.

That gap between "asked" and "answered" is the heart of this book. A modern processor executes an enormous number of instructions per second. A network round trip to a server, even one on a fast connection, takes a noticeable fraction of a second. During a single 0.25-second wait, a processor running at a few gigahertz could have executed hundreds of millions of clock cycles of work. A program that makes twenty such requests one after another spends nearly all of its life waiting, and only a sliver of it working.

We will use two plain words for these two states throughout the book:

- **Waiting** means the program cannot continue until something outside the processor finishes: a network reply, a disk read, a database query, a timer, a child process. While it waits, the processor could do other work.
- **Working** means the processor is busy executing your code: looping, calculating, parsing, compressing. The only way to finish working sooner is to do less work, do it more cleverly, or use more processor cores at the same time.

Concurrency is the art of not wasting the waiting. If a program has twenty independent requests to make and each one spends most of its time waiting for a reply, there is no reason to wait for them one at a time. You can send all twenty, wait for all of them together, and handle each reply as it arrives. The total time drops from "the sum of all the waits" toward "the longest single wait". That is the entire trick, and the rest of this book is about doing it correctly, safely, and measurably.

A kitchen is a useful picture. A cook who stares at a pot until it boils before chopping anything serves dinner late. A real cook chops while the water heats. Nothing about the cook got faster; the meal is ready sooner because the waiting overlaps. That is concurrency. Hiring a second cook who chops at the same instant as the first is a different thing called parallelism, and we will separate those two ideas carefully in Chapter 2.

Why this matters more than it used to

Programs today talk to other programs constantly: web APIs, storage buckets, databases, internal services. Each call is mostly waiting. Done one after another, the waits add up and users notice; done concurrently, they overlap and the program feels responsive.

The second reason is that machines now gain cores rather than much faster single cores. The machine we used for this book reports 32 logical CPUs. A program that uses one core for heavy computation leaves the rest idle, and Python has its own rules about using them, which we explore in Chapter 3.

Meet Gullwing

Every example in this book serves one small, realistic project. Gullwing is a fictional team building a price tracker. The idea is simple: watch a handful of products across several online stores, find the best in-stock price for each product, and send an alert when a price drops to or below the price a user is willing to pay.

Three people appear now and then. **Nadia Brennan** founded Gullwing and writes the requirements. **Tomás Ruiz** is the backend developer who has to make the tracker fast; when we narrate a decision, think of Tomás as standing in for you. **Wren Okafor** runs operations and cares about Gullwing being a polite client of other people's servers and shutting down cleanly. They are characters in the story, not voices in the book.

Nadia's first requirement fits on an index card:

1. Poll four stores for five products: twenty price checks per run.
2. For each product, find the lowest price among the stores that have it in stock.
3. Compare that best price with the watchlist target. If the best price is at or below the target, send an alert.
4. Print a short report. Do it again every minute.

The first version Tomás writes checks the stores one at a time. It works, and it is correct, and it takes almost five seconds per run. Nadia asks whether it can be faster, because next quarter she wants forty stores and two hundred products. That question drives the whole book.

The stores

Gullwing never touches a real store. The sample project includes a fake store server, a tiny local web server that pretends to be five stores, each with a fixed, deliberate delay before it answers. The delays are what make the examples interesting: they stand in for real network latency, and because they are fixed, you can predict the timings with simple arithmetic and then check the prediction against a real run.

Slug	Store name	Delay per request
`driftwood`	Driftwood Supply	0.10 s
`harborline`	Harborline Outfitters	0.15 s
`pinecrest`	Pinecrest Market	0.25 s
`saltmarsh`	Saltmarsh Goods	0.40 s
`fogbank`	Fogbank Electronics	2.50 s (not polled by default)

The first four make up `DEFAULT_STORES`, the stores Gullwing polls on every run. Fogbank is the slow one. It is cheap, but it takes two and a half seconds to answer, which makes it perfect for the timeout and cancellation chapters later on. The order of the stores matters: it is the order in which Gullwing sends requests, and it breaks ties when two stores offer the same price.

The products and the watchlist

Gullwing tracks five products, identified by a SKU (stock keeping unit). Prices are always stored as integer cents, so $84.99 is the integer `8499`. Floating-point dollars cause rounding surprises, and price trackers that compare prices for equality cannot afford them.

SKU	Product	Watch target	Best in-stock price
GW-101	Trailhead 40L Backpack	$85.00	$84.99 Harborline
GW-102	Copper Pour-Over Kettle	$40.00	$42.95 Harborline
GW-103	Stillwave ANC Headphones	$185.00	$184.99 Pinecrest
GW-104	Larkspur 75 Mechanical Keyboard	$120.00	$119.99 Pinecrest
GW-105	Emberline 12-inch Cast Iron Skillet	$33.00	$34.99 Driftwood

Three products are at or below target, so a correct run produces exactly three alerts: GW-101, GW-103, and GW-104. Two details are worth noticing now because they will catch a careless implementation later. Saltmarsh sells the backpack for $79.99, cheaper than Harborline, but it is out of stock, so it does not count. Pinecrest sells the skillet for $32.99, which would be under the $33.00 target, but it is also out of stock, so GW-105 does not alert. Out-of-stock offers never win. Not every store carries

every product either: Pinecrest does not sell the kettle and Harborline does not sell the keyboard. Those requests come back as "not found", so a full run makes 20 requests and finds 18 offers, 16 of them in stock.

> **NOTE**
>
> The prices, stores, and people in Gullwing are fictional. Every request in this book goes to a server running on your own machine at `127.0.0.1`. Nothing in the examples contacts a real store or any public service, so you can run them offline and as often as you like.

Setting Up the Sample Project

You need **Python 3.12 or later**. Every example in this book was run on CPython 3.12 on Windows 11, and everything we show works on macOS and Linux as well. A few features that arrived in Python 3.13 and 3.14 are mentioned where they matter, always labeled as such and never as the only way to do something.

The sample project is a normal Python project with two packages. `gullwing` is the price tracker. `fakestore` is the fake store server. Here is the layout of the finished project; we will build most of the `gullwing` modules step by step, chapter by chapter, so do not worry if the list looks long.

Project layout

```
sample/
  pyproject.toml
  fakestore/   __init__.py  __main__.py  data.py  server.py
  gullwing/    __init__.py  __main__.py  models.py  catalog.py  config.py
               timing.py  fetch_sync.py  fetch.py  aggregate.py  alerts.py
               board.py  ratelimit.py  breaker.py  pipeline.py  storage.py
               analysis.py  poll.py  scheduler.py
  tests/       conftest.py  test_aggregate.py  test_fetch.py  test_extras.py
```

Create a virtual environment in the project folder and install the three runtime libraries the sample uses: `requests` for the blocking version, `httpx` for the async client, and `aiohttp`, which powers the fake store server. For the testing chapter you will also want `pytest` and `pytest-asyncio`. You already know how to create a virtual environment and install packages, so we will only list the command once.

Terminal

```
python -m pip install httpx aiohttp requests pytest pytest-asyncio
```

On Windows, use `py` if `python` is not on your PATH. If you have several versions installed, `py -3.12` selects Python 3.12 explicitly.

Starting the fake store server

The fake store server is an ordinary Python module. Run it from the project folder in its own terminal window, because it keeps running until you stop it with Ctrl+C. The default port is 23500. In this chapter we use 23501 so that our examples never collide with a server you might already have running; any free port works.

Terminal

```
python -m fakestore --port 23501
```

Output

```
Fake stores on http://127.0.0.1:23501 (flaky=off, strict-limits=off)
```

The two "off" switches are for later: `--flaky` makes Saltmarsh fail the first request for each product (Chapter 18), and `--strict-limits` refuses clients that send too many requests to one store at once (Chapter 17).

Check that it is alive from a second terminal. You could use a browser or `curl`, but a Python one-liner works everywhere:

Terminal

```
python -c "import urllib.request as u; print(u.urlopen(
    'http://127.0.0.1:23501/health').read().decode())"
```

Output

```
{"status": "ok", "stores": 5}
```

(Type the command on one line; we wrapped it to fit the page.) Asking a store for one product returns the offer as JSON. This is the request Gullwing makes twenty times per run:

Terminal

```
python -c "import urllib.request as u; print(u.urlopen(
    'http://127.0.0.1:23501/stores/harborline/products/GW-101').read().decode())"
```

Output

```
{"store": "harborline", "sku": "GW-101", "name": "Trailhead 40L Backpack",
"price_cents": 8499, "currency": "USD", "in_stock": true}
```

This command took a moment longer than the health check. That pause is the 0.15-second Harborline delay, and it is what this book is about.

> **TIP**
>
> Keep the fake store server in its own terminal window or tab for the whole session and leave it running while you work through a chapter. Stop it with Ctrl+C when you are done. If you see "address already in use", another program owns that port; choose a different port number rather than hunting down and killing a process you did not start.

The Promise: One Poll, Three Timings

The finished sample includes a command-line interface. You will build every piece of it, but it is worth running the finished product once now, so that you know where we are going. The `poll` command fetches all twenty prices, picks the best in-stock offer per product, checks the watchlist, posts any alerts to the fake server, and prints a report. The `--mode` option picks how the twenty requests are made.

Mode 1: one request at a time

Terminal

```
python -m gullwing poll --port 23501 --mode sync
```

Output

```
Gullwing poll (sync): 18 offers from 4 stores
SKU     Product                                  Best  Store
GW-101  Trailhead 40L Backpack                  $84.99  Harborline Outfitters
GW-102  Copper Pour-Over Kettle                 $42.95  Harborline Outfitters
GW-103  Stillwave ANC Headphones               $184.99  Pinecrest Market
GW-104  Larkspur 75 Mechanical Keyboard        $119.99  Pinecrest Market
GW-105  Emberline 12-inch Cast Iron Skillet     $34.99  Driftwood Supply
ALERT GW-101 Trailhead 40L Backpack: $84.99 at Harborline Outfitters (target $85.00)
ALERT GW-103 Stillwave ANC Headphones: $184.99 at Pinecrest Market (target $185.00)
ALERT GW-104 Larkspur 75 Mechanical Keyboard: $119.99 at Pinecrest Market (target $120.00)
3 alerts. Elapsed: 4.74 s
```

The report is correct: 18 offers, the right best prices, and exactly the three alerts we predicted. It took about 4.7 seconds on our machine. We can predict that number without running anything. Each of the five products needs one request to each of the four stores, and the store delays add up to 0.10 + 0.15 + 0.25 + 0.40 = 0.90 seconds per product. Five products times 0.90 seconds is **4.50 seconds** of pure waiting. The remaining couple of tenths of a second is everything else: starting Python, opening connections, parsing JSON, printing.

In other words, this program spends roughly 95 percent of its running time waiting for replies. The processor is nearly idle. That is the clearest possible sign that concurrency will help.

Mode 2: four threads

Terminal

```
python -m gullwing poll --port 23501 --mode threads
```

Output

```
Gullwing poll (threads): 18 offers from 4 stores
SKU     Product                                  Best  Store
GW-101  Trailhead 40L Backpack                $84.99  Harborline Outfitters
...
3 alerts. Elapsed: 1.34 s
```

Same report, same three alerts (we cut the middle lines). This time the program used a pool of four threads, so up to four requests were waiting at the same moment. It finished in about 1.3 seconds. Nothing else changed: the same `requests` library, the same blocking calls, the same server.

Mode 3: asyncio

Terminal

```
python -m gullwing poll --port 23501 --mode async
```

Output

```
Gullwing poll (async): 18 offers from 4 stores
SKU     Product                                  Best  Store
GW-101  Trailhead 40L Backpack                $84.99  Harborline Outfitters
...
3 alerts. Elapsed: 1.54 s
```

The async version runs on a single thread, with up to eight requests in flight at once (its default limit), and finished in about 1.5 seconds on this run. Across several runs on our machine it landed between about 1.3 and 1.8 seconds. That includes posting the three alerts, which the blocking modes skip, and creating the async HTTP client. With a higher limit (`--limit 20`) it came in at about 1.3 seconds. Our machine was busy with other work during these runs, which inflates the overhead; the pure latency arithmetic for eight requests in flight at once is 0.80 seconds, and a quieter machine lands closer to that, plus the client's setup cost.

Do not read these numbers as a race between threads and asyncio; the gap between the two concurrent modes comes from their settings and setup costs. What matters is the shape: both concurrent versions are several times faster than the sequential one, simply because they stopped waiting for one reply before asking the next question.

> **WARNING**
>
> Timings in this book are measurements from our test machine, labeled "about". Your numbers will differ with your hardware, operating system, background load, and Python version. Treat them as illustrations of a shape, never as benchmarks. Chapter 25 shows how to measure your own programs properly, including repeated runs and medians.

Where the floor is

How fast could a poll possibly be? If all twenty requests were sent at the same instant, the run could not finish before the slowest single reply arrived: Saltmarsh's 0.40 seconds. That is the **latency floor**. No amount of concurrency gets below it, because concurrency overlaps waits; it does not shorten them. Real runs land above the floor because of client overhead, connection setup, and the limits we deliberately impose so that Gullwing does not flood the stores. Wren will have strong opinions about those limits in Chapter 17.

This is why Nadia's growth plan is realistic. Eight thousand requests (forty stores times two hundred products) at a few tenths of a second each would take over half an hour sequentially. Concurrency changes the question from "how many requests" to "how many at once".

The Three Models at a Glance

Python gives you three main ways to do more than one thing at a time. Each has its own strengths, costs, and rules, and the first half of the book is about understanding them well enough to choose. Here is a first look.

Threads

A thread is an independent path of execution inside your process. All threads share the same memory: the same objects, the same module globals, the same open files. The operating system decides when each thread runs. When a thread makes a blocking call, such as waiting for a network reply, the operating system lets another thread run. Threads are the oldest tool and work with almost every library, including blocking ones like `requests`. That is exactly what Mode 2 did.

Threads have two big catches. First, because they share memory and can be switched at almost any moment, two threads can step on each other's data. Those bugs are called race conditions, and Chapter 5 shows real ones. Second, in standard CPython only one thread executes Python code at any instant, because of the global interpreter lock (the GIL). Threads overlap waiting beautifully, but they do not make pure-Python computation faster. Chapter 3 explains why, and what the newer free-threaded builds of Python change.

Processes

A process is a separate running copy of the Python interpreter with its own memory. Separate processes can run Python code on different processor cores at the same time, so processes are the classic way to speed up heavy computation in Python. The price is isolation: processes do not share objects, so data has to be copied between them, usually by pickling. Starting a process also costs far more than starting a thread. Chapters 8 through 10 cover processes, the `multiprocessing` module, and process pools, along with the Windows and macOS rules about the `if __name__ == "__main__":` guard.

asyncio

With `asyncio`, a single thread runs an **event loop** that juggles many tasks. Each task is a coroutine, written with `async def`, that voluntarily pauses at every `await` while it waits for something. While one task is paused, the loop runs another. Nothing is interrupted unexpectedly: switches happen only at `await` points you can see in the code. That makes many shared-state bugs impossible and makes it practical to have thousands of waits in flight. The price is that everything on the path must cooperate. A single blocking call inside a coroutine stalls every task on the loop, and you need async-aware libraries such as `httpx` or `aiohttp`. Parts III and IV are about asyncio.

A timeline picture

Here is one product's four requests drawn three ways. Each character is roughly 0.05 seconds of waiting.

Timeline

```
Sequential: one request at a time (0.90 s for one product)
driftwood   ==
harborline    ===
pinecrest        =====
saltmarsh             ========
            |----|----|----|----|----|
            0   0.2  0.4  0.6  0.8  1.0 s

Concurrent (threads or asyncio): all four wait together (0.40 s)
driftwood   ==
harborline  ===
pinecrest   =====
saltmarsh   ========
            |----|----|
            0   0.2  0.4 s

Parallel (processes): useful when each bar is WORK, not waiting
core 1      ########
core 2      ########
core 3      ########
core 4      ########
```

The first two diagrams show the same requests. The only difference is whether each one waits for the previous one to finish. The third diagram is a different situation: the bars are computation, and the only way to finish four computations in the time of one is to run them on four cores at once. Telling these two situations apart is the most important skill in this field, and Chapter 2 gives you the tools to do it.

Model	Good at	Main cost	Chapters
Threads	Overlapping waits with blocking libraries	Shared-memory races; GIL limits CPU work	4–7
Processes	Pure-Python CPU work on many cores	Startup and copying (pickling) costs	8–10
asyncio	Many concurrent waits on one thread	Needs async libraries; blocking calls stall everything	11–20
Hybrid	I/O and CPU work in one program	More moving parts	10, 21, 26

> **NOTE**
> You will sometimes read that asyncio is "faster" than threads, or the reverse. For waiting-heavy work, both can overlap waits well; the differences show up in how many concurrent waits you can afford, how cancellation and timeouts work, and how easy the code is to get right. We compare the models on those terms in Chapter 26, not with a single speed number.

Concurrency Is Not Free

If concurrency only made programs faster, everyone would use it everywhere. It also makes programs harder to reason about. A sequential program does step one, then step two, then step three, and the order is visible on the page. A concurrent program does several things whose steps interleave in ways that can change from run to run. That leads to a family of problems you may never have met in sequential code:

- **Race conditions.** Two threads read a shared value, both change it, and one change is lost. The program is correct most of the time and wrong occasionally, which is the worst kind of bug.
- **Deadlocks.** Two threads each hold a lock the other needs, and both wait forever.
- **Blocked event loops.** One careless blocking call inside a coroutine freezes hundreds of other tasks.
- **Orphaned work.** A task keeps running after the code that started it has given up, or an error in a background task disappears silently.
- **Overload.** A program that can send a thousand requests at once might actually do it and get blocked or rate-limited by the server it depends on.

- **Shutdown.** Stopping a concurrent program cleanly, without losing in-flight work or leaving connections open, takes deliberate design.

Our approach is to show the failure first: a lost update, a frozen loop, a server answering "too many requests", running for real, and only then the fix. Seeing a failure once makes it easy to recognize later.

We also lean on **structured concurrency**, the idea that concurrent work should have a clear owner and a clear lifetime, just like a function call. Code that starts tasks should wait for them, handle their errors, and cancel them if it gives up. Python 3.11 added `asyncio.TaskGroup` for exactly this, and Chapter 13 is built around it. Even in the threads chapters, we write code where every thread we start is joined, and every pool we create is shut down.

When not to bother

Concurrency is a tool, not a virtue. A script that runs once a day and finishes in two seconds does not need it. A program whose time is all spent in one database query will not get faster by wrapping that query in a task. A computation dominated by a NumPy call that already uses several cores will gain little from more processes. The first rule of this book, which Chapter 2 makes concrete, is **measure first**: find out whether your program is waiting or working, and where, before you change anything.

Gullwing itself will need both cures. Nadia's second feature is a "price noise" analysis that crunches a long simulated price history for each product. That is working, not waiting, and overlapping waits will not speed it up at all. Chapter 2 measures both kinds of slowness side by side.

How This Book Is Organized

The book has six parts, and each builds on the previous ones. Gullwing grows from a slow script into a polite, resilient, tested, concurrent service as we go.

- **Part I — Foundations** (Chapters 1–5). Waiting versus working, concurrency versus parallelism, and how to measure. The GIL and what free-threaded Python changes. Threads, and how to keep shared data correct.
- **Part II — Threads and Processes** (Chapters 6–10). Queues, events, and semaphores for coordinating threads. `concurrent.futures` and thread pools. Multiprocessing, sharing state between processes, and how to choose between threads and processes for CPU work.
- **Part III — asyncio Fundamentals** (Chapters 11–15). Coroutines and the event loop from first principles, tasks, structured concurrency with `TaskGroup`, timeouts and cancellation, and async iteration and context managers.
- **Part IV — Async I/O in Practice** (Chapters 16–20). Async HTTP clients, limiting concurrency and rate limiting, retries and circuit breakers, queue-based pipelines, and a look inside the fake store server itself.