The ‘t’ Build Playbook !

A practical guide to migrating to free-threaded Python

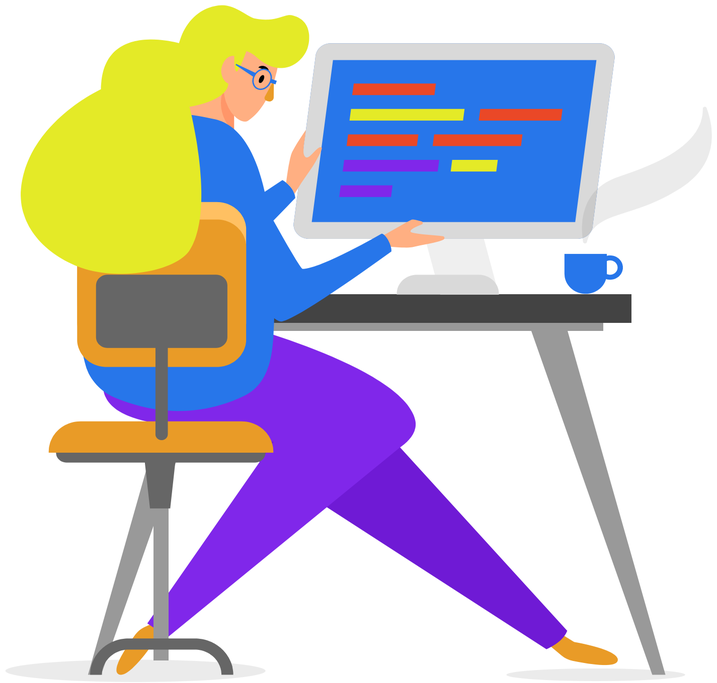
Swaraj Pande
Software Engineer (L2) @ Red Hat

Goodbye GIL: Exploring Free-threaded Python 3.13 and Beyond
by Adarsh Divakaran

Talk Link: https://2025.pycon.jp/ja/timetable/talk/KLCT8T

Youtube Link: https://www.youtube.com/watch?v=Qa41zUOzZaU

Agenda (of problems)



1. Upgraded but nothing changed
2. GIL comes back with no crash
3. A C / Cython package isn't ready
4. Counts and caches go wrong
5. More threads make it slower
6. One-thread code got worse
7. CI still tests the old Python

What are we migrating to?

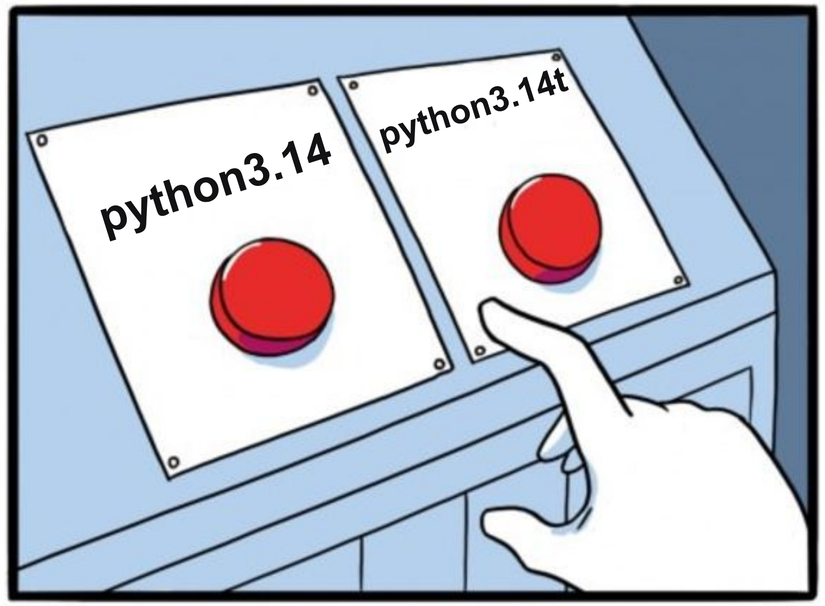
Python **3.14** has a GIL by default.

The free-threaded (GIL disabled) one is a second interpreter **“python3.14t”**.

This is supported now (no longer experimental).

However, this is not the default.

You can opt in. You can run both !



Problem 1: “We moved to 3.14” and the GIL is still there

What you see: Version says 3.14. Threads still don’t use extra cores

Why: You installed the default build. Or you installed ‘t’ but never checked after imports.

```
$ python3.14t -VV
Python 3.14.7 free-threading build (main, Aug  5 2026, 10:29:49) [Clang 21.0.0 (clang-2100.1.1.101)]
```

```
$ python3.14t
Python 3.14.7 free-threading build (main, Aug  5 2026, 10:29:49) [Clang 21.0.0 (clang-2100.1.1.101)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys, sysconfig
>>> sysconfig.get_config_var("Py_GIL_DISABLED")
1
>>> sys._is_gil_enabled()
False
```

1 means this binary **can** turn off the GIL !

False means GIL is **off right now** !

Problem 2: The GIL comes back and nobody pages

What you see: Throughput looks like the old GIL box. No crash. A warning in the logs.

Why: Some C module never said “I can run without the GIL”. Python turns the GIL **back on** for the whole process and continues.

Fix:

- After every import, if GIL is on and you expected if off -> crash the process (CI and prod)
- Do not “fix” it with PYTHON_GIL=0 in production.
  That runs unmarked C unsafely.

```python
# Add all the project imports in this array !
PRODUCTION_IMPORTS = ["json", "threading", "math", "ssl", "ctypes", "hashlib", "array",]
for name in PRODUCTION_IMPORTS:
    importlib.import_module(name)
    if sys._is_gil_enabled():
        raise RuntimeError(f"{name} turned the GIL back on")
```

Problem 3: A package with C inside is not ready 😭

What you see: Import probe fails, or pip installed a normal cp314 wheel instead of a cp314t wheel into a ‘t’ virtual environment.

Why: Native code is different binary. No ‘t’ when -> not ready. Copy-pasting “compatible” in Cython does **not** add locks.

Fix:

If you have …	**Do this**
Pure Python	Fine. Still test with threads
A cp314t wheel	Pin that version
Only a normal wheel	Rebuild, wait, or replace
Your own Cython / C	Declare support after multithreaded tests
A vendor blob you cannot rebuild	Assume it turns the GIL on ← Rare !

See what is ready: py-free-threading.github.io/tracking

Problem 4: Numbers are wrong (the GIL was a fake lock)

What you see: Counters too low, cache misses, duplicate work. Fine on old Python. Wrong on ‘t’.

Why: For example, counter += 1 is not a one step. Two threads can both read 0 and both write 1. The GIL **made** that lock safe. There was no locks implemented in your code. Once GIL gone, they emerged !

Fix: Use a real Lock, a queue, or don’t share.

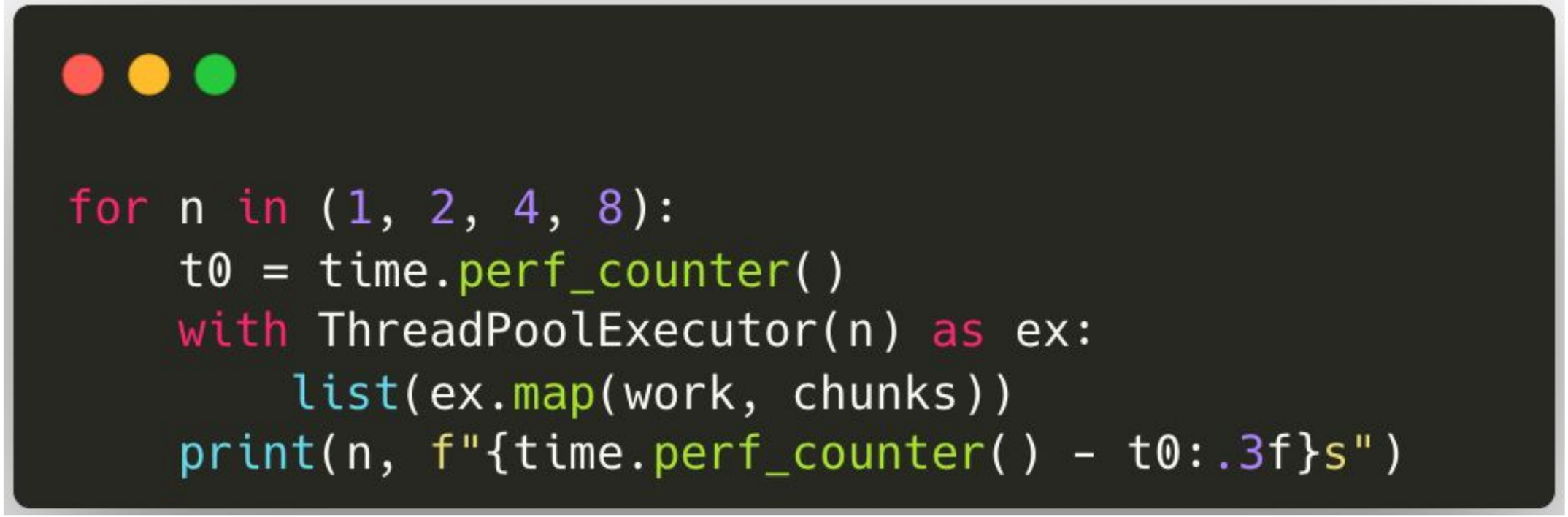
Problem 5: More threads, worse speed ! 💀

What you see: 2 cores help a bit. 8 cores are flowers. CPU looks idle.

Why: Every thread touching the same Python object (like config or common dictionary) fights over the same CPU cache line (often the refcount). That is **false sharing, not a race.**

Fix:

- Stop sharing the hot object. Copy config per thread. Split the work list.
- Sweep 1 / 2 / 4 / 8 workers. Stop when the next doubling does not cut wall time.


for n in (1, 2, 4, 8):
t0 = time.perf_counter()
with ThreadPoolExecutor(n) as ex:
list(ex.map(work, chunks))
print(n, f"{time.perf_counter() - t0:.3f}s")

Problem 6: Single-thread code got slower and fatter !

What you see: A one-thread API or CLI is 5-10% slower. Memory is higher.

Why: The ‘t’ build pays extra sync even when you don’t use threads. Objects are a bit bigger. Memory is freed later.

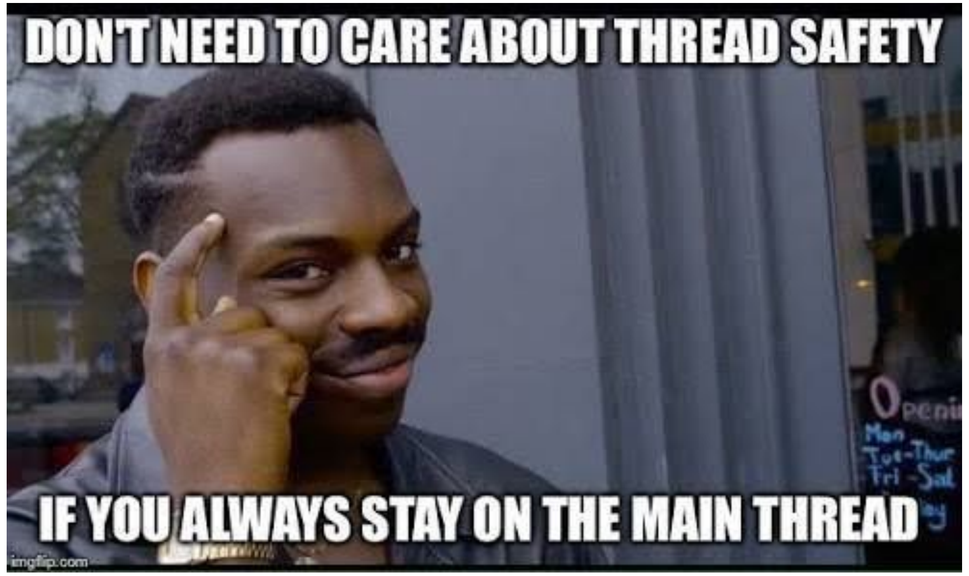
Fix:

- Measure **one thread** and **many threads** versions before you ship
- Use t on the **CPU worker**. Keep the latency path on default 3.14 if the tax isn’t paid back
- Don’t put API and worker in one “simple” image “because 3.14”.

Problem 6: Single-thread code got slower and fatter !

What you see: A one-thread API or CLI is 5-10% slower. Memory is higher.

Why: The ‘t’ build pays extra sync even when you don’t use threads. Objects are a bit bigger. Memory is freed later.



Fix:

- Measure **one thread** and **many threads** versions before you ship
- Use t on the **CPU worker**. Keep the latency path on default 3.14 if the tax isn’t paid back
- Don’t put API and worker in one “simple” image “because 3.14”.

Problem 7: CI only runs python3.14 !

What you see: Green pipeline Hurray !
Production t is a different program.

Why: Default 3.14 and 3.14t are two ABIs. Tests on one do not cover the other.

Fix: Add this (or equivalent) in your CI YAML
python: [“3.14”, “3.14t”]

On the t job:

- python -VV contains free-threading
- pip used cp314t wheels, not cp314
- Import probe must keep GIL off
- A few tests that actually run on several threads
- (nightly) one-thread vs N-thread benchmark

Thank you!

Connect

 @swarajpande05

 /swarajpande5

 @swarajpande5

QnA