

# How Not to Write Async Python (and What to Do Instead)

Juan Pablo Vanegas  
Backend Engineer @ Loka



whoami



jpvanegasc



LOK1

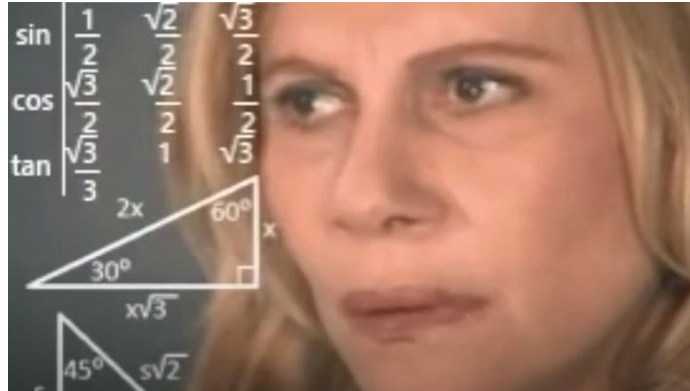
This talk aims to help you write faster code with less bugs in less time by:

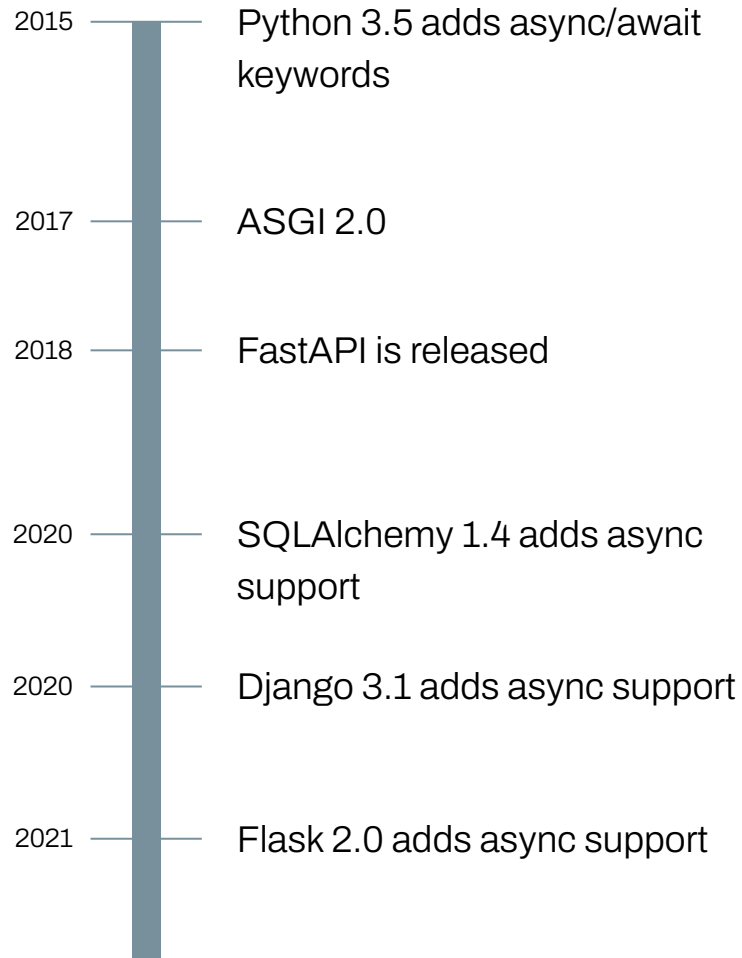
Triggering the pattern-recognizing human brain to identify async code smells and have a couple of tools at hand to solve them

It's not:

- Deep dive on the event loop implementation
- Comparison between event loops (asyncio, uvloop) or backends (asyncio, trio)
- Philosophical discussions on programming techniques
- In-depth python core concepts (GIL and so on)

# The Async Hype: Where Are We Now?





“Faster execution because your program doesn’t stay idle while waiting for a result”

# Sync vs. Async vs. Parallel: What's the Difference?



Let's make frozen pizza and a salad for dinner:

Option 1:

1. Open pizza box
2. Put pizza in oven
3. Wait 10 min
4. Put pizza in plate
5. Cut vegetables
6. Mix vegetables
7. Put vegetables in plate
8. Eat

Option 2:

1. Open pizza box
2. Put pizza in oven
3. Put timer for 10 min
4. Cut vegetables
5. Mix vegetables
6. Put vegetables in plate
7. Put pizza in plate
8. Eat

Option 3:

0. Call a friend

You:

1. Open pizza box
2. Put pizza in oven
3. Wait 10 min
4. Put pizza in plate

Them:

1. Cut vegetables
2. Mix vegetables
3. Put vegetables in plate

Eat

Now in programming terms:

Two kinds of waiting: I/O-bound and CPU bound:

- I/O-bound: waiting on *something else* to give the result
- CPU-bound: waiting on your own calculations to finish

Async is more suited for I/O bound, parallel for CPU-bound:

- Parallel adds overhead.
- Asynchronous has limited processing power available.



# How the Event Loop *Actually* Works



The event loop manages the execution of **asynchronous operations** by continuously checking a queue for **tasks that are ready to be executed**, allowing the program to run **non-blocking code** while still processing events.

- **async operations**: operations that depend on *something else* to finish
- **tasks ready to be executed**: tasks that haven't started waiting or that have already finished waiting
- **Non-blocking code**: code that doesn't need to wait

```
1  class Task:
2      def __init__(self, coro):
3          self._coro = coro
4          self.value = None
5
6      def run(self):
7          return self._coro.send(self.value)
8
9  class GeneratorEventLoop:
10     def __init__(self):
11         # Yes I could use deque or something similar, but
12         # because I'm not removing the tasks I'd rather keep it simple
13         self.ready = []
14         self.next = []
15
16     def run_almost_forever(self):
17         while self.ready or self.next:
18             for task in self.ready:
19                 try:
20                     task.run()
21                     self.next.append(task)
22                 except StopIteration as stop:
23                     task.value = stop.value
24             self.ready = self.next
25             self.next = []
26
27     def schedule(self, coro):
28         task = Task(coro)
29         self.next.append(task)
30         return task
31
```

```
1 ✓ def _almost_a_coroutine(max):  
2     print("started")  
3 ✓     for i in range(max):  
4         print(f"doing stuff before {i=}")  
5         yield i * spam(2)  
6     print("one more yield")  
7     yield  
8     print("done")  
9     return max  
10  
11 gen_loop = GeneratorEventLoop()  
12 t = gen_loop.schedule(_almost_a_coroutine(3))  
13 gen_loop.run_almost_forever()  
14 print(f"{t.value=}")
```

```
started  
doing stuff before i=0  
doing stuff before i=1  
doing stuff before i=2  
one more yield  
done  
t.value=3
```

```
1 ✓ def _almost_a_coroutine(id):  
2     print(f"Starting {id=}")  
3     yield spam(2)  
4     print(f"Finished {id=}")  
5  
6 gen_loop.schedule(_almost_a_coroutine(1))  
7 gen_loop.schedule(_almost_a_coroutine(2))  
8 gen_loop.schedule(_almost_a_coroutine(3))  
9 gen_loop.run_almost_forever()
```

```
Starting id=1  
Starting id=2  
Starting id=3  
Finished id=1  
Finished id=2  
Finished id=3
```

# The Problem with Awaiting Everywhere



`await` tells the interpreter it should wait for something to complete.

- Do you *need* the result from the coroutine you're calling?
- Does the coroutine perform something that has to be done as a standalone request.

If you didn't answer yes to any of these questions, you might not want to await.

- Maybe you need *several* results (e.g., a select and a count query)
- Maybe the task can be completed eventually after you schedule

```
1 ✓ async def eggs(id):  
2     print (f"starting {id}")  
3     await async_spam(2)  
4     print(f"finished {id}")  
5  
6 ✓ async def _run():  
7     await eggs(1)  
8     await eggs(2)  
9     await eggs(3)  
10  
11 await _run()
```

```
starting id=1  
finished id=1  
starting id=2  
finished id=2  
starting id=3  
finished id=3
```

# Fixing Async Code: The Right Tools

(to get you started)



## Common code smells:

- Defining a coroutine without awaiting anything inside → change `async def` to `def`
- Calling a blocking function inside a coroutine (using a library that doesn't support async) → Find a library that does, or wrap the sync function (`asyncio.run_in_executor`, `anyio.to_thread.run`, etc.)
- Awaiting inside loops → use `gather`
- Multiple awaits one after another → use `gather` or `TaskGroup`



```
1 ✓ async def _shouldnt_be_a_coroutine(max_val):
2     sum = 0
3 ✓     for i in range(max_val):
4         sum += 0.1 * i
5     return sum
6
7 ✓ async def all_the_bad_things():
8     sum_result = await _shouldnt_be_a_coroutine(2**10)
9     blocking_result = spam(1) # async_spam is available!
10
11     result_1 = await async_spam(0.5)
12     result_2 = await async_spam(0.6)
13
14     values = []
15 ✓     for i in range(10):
16         value = await async_spam(0.1 * i)
17         values.append(value)
18
19     return sum_result + blocking_result + result_1 + result_2 + sum(values)
20
21 _start = time.time()
22 _result = await all_the_bad_things()
23 _end = time.time()
24 print(f"done in {_end - _start:.5f} seconds")
25 print(f" {_result=}")
```

```
done in 6.61333 seconds
_result=52384.2
```

```
1  def _compute_heavy(max_val):
2      sum = 0
3      for i in range(max_val):
4          sum += 0.1 * i
5      return sum
6
7  def pure_sync():
8      sum_result = _compute_heavy(2**10)
9      blocking_result = spam(1)
10
11     result_1 = spam(0.5)
12     result_2 = spam(0.6)
13
14     values = []
15     for i in range(10):
16         value = spam(0.1 * i)
17         values.append(value)
18
19     return sum_result + blocking_result + result_1 + result_2 + sum(values)
20
21 _start = time.time()
22 _result = pure_sync()
23 _end = time.time()
24 print(f"done in {_end - _start:.5f} seconds")
25 print(f"{_result=}")
```

```
done in 6.64935 seconds
_result=52384.2
```

Your new best friends:

`asyncio.gather(*coroutines)`

→ Use it to immediately run several coroutines (e.g. several requests to an API service).

`asyncio.create_task(coroutine)`

→ Use it to *schedule* a coroutine for execution. Might be done now, might be done later.

`asyncio.TaskGroup()` (Introduced in 3.11)

→ Similar to `gather`. Use it to easily hold a reference to the tasks! (coroutines sent to `gather` or to `create_task` might be garbage collected and mysteriously disappear if not properly referenced)

→ Catch exceptions with the `except*` syntax

```
1 ✓ async def bad_loop():
2   for _ in range(100):
3     await asyncio.sleep(0.1)
4
5 ✓ async def good_gather():
6   tasks = [asyncio.sleep(0.1) for i in range(100)]
7   await asyncio.gather(*tasks)
8
9   _start = time.time()
10  await bad_loop()
11  _end = time.time()
12  print(f"bad done in {_end - _start:.5f} seconds")
13
14  _start = time.time()
15  await good_gather()
16  _end = time.time()
17  print(f"good done in {_end - _start:.5f} seconds")
```

```
bad done in 10.10158 seconds
good done in 0.10251 seconds
```

```
1 tasks = set() # for keeping a reference
2
3 async def something_i_need_now():
4     await async_spam(0.1)
5     print("here you go")
6
7 async def something_that_can_run_later():
8     print("starting later")
9     await async_spam(1)
10    print("did the thing!")
11    return 0
12
13 def mark_done(task):
14     global tasks
15     tasks.remove(task)
16     print(f"Finished {task.get_name()}")
17
18 async def run():
19     task = asyncio.create_task(something_that_can_run_later(), name="later_task")
20     task.add_done_callback(mark_done)
21     tasks.add(task)
22
23     await something_i_need_now()
24
25     while not task.done():
26         await asyncio.sleep(0.5)
27
28     if task.done():
29         print(f"Result={task.result()}")
30
31     await run()
```

```
starting later
here you go
did the thing!
Finished later_task
Result=0
```

```
1  ✓ async def get_data():
2      print("getting data...")
3      await async_spam(1)
4      print("got data")
5      return [0, 1, 2, 3, 4, 5]
6
7  ✓ async def get_size():
8      print("getting size...")
9      await async_spam(0.1)
10     print("got size")
11     return 6
12
13  ✓ async def bad_awaits():
14     data = await get_data()
15     size = await get_size()
16     print(f"{data=}, {size=}")
17
18  ✓ async def task_group():
19  ✓     async with asyncio.TaskGroup() as tg:
20         t1 = tg.create_task(get_data())
21         t2 = tg.create_task(get_size())
22
23         data = t1.result()
24         size = t2.result()
25
26         print(f"{data=}, {size=}")
27
```

```
1  # Not really async, see?
2  await bad_awaits()
```

```
getting data...
got data
getting size...
got size
data=[0, 1, 2, 3, 4, 5], size=6
```

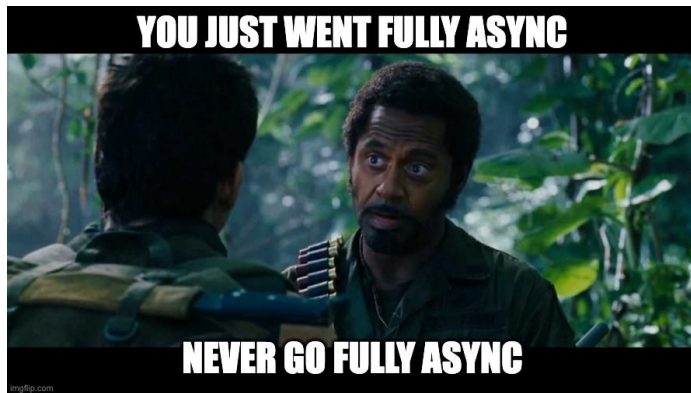
```
1  # Async, more manageable, without the garbage collection risk
2  await task_group()
```

```
getting data...
getting size...
got size
got data
data=[0, 1, 2, 3, 4, 5], size=6
```

```
1 def _not_actually_a_coroutine(max_val):
2     sum = 0
3     for i in range(max_val):
4         sum += 0.1 * i
5     return sum
6
7 async def _actually_a_coroutine(max_val):
8     return await asyncio.gather(*[async_spam(0.1 * i) for i in range(max_val)])
9
10 async def all_the_good_things_taskgroup():
11     value = _not_actually_a_coroutine(2**10)
12
13     async with asyncio.TaskGroup() as tg:
14         long_result = tg.create_task(async_spam(1))
15         result_1 = tg.create_task(async_spam(0.5))
16         result_2 = tg.create_task(async_spam(0.6))
17         values = tg.create_task(_actually_a_coroutine(10))
18
19     return (
20         value
21         + long_result.result()
22         + result_1.result()
23         + result_2.result()
24         + sum(values.result())
25     )
26
27 _start = time.time()
28 _result = await all_the_good_things_taskgroup()
29 _end = time.time()
30 print(f"done in {_end - _start:.5f} seconds")
31 print(f" {_result}")
```

```
done in 1.00144 seconds
_result=52384.2
```

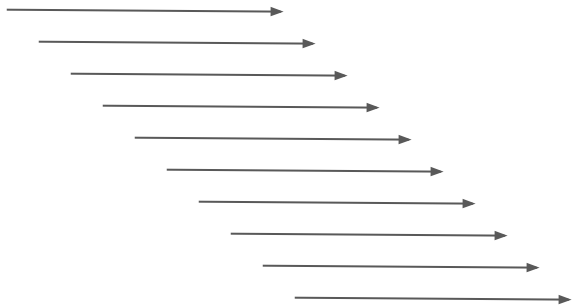
# Concurrency Limits: When to *Not* Go Fully Async



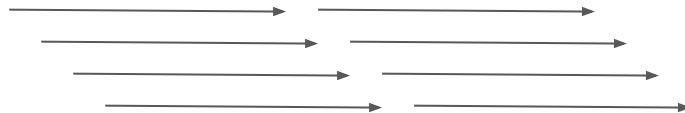


Did you (like me) get excited about `gather` and sent thousands of requests to an external API? You might just have crashed someone else's service (or, if it was your DB, your own) :(

Sometimes, you *really* just want a couple of async requests made at the same time: Introducing `asyncio.Semaphore`



Send everything right away



Wait for some of the tasks to continue before sending more

```
1 ✓ class MyCoolService:
2     max_size = 10
3
4 ✓     def __init__(self):
5         self.current_services = set()
6
7 ✓     async def my_slow_action(self, id):
8 ✓         if len(self.current_services) > self.max_size:
9             raise RuntimeError("You broke my service :(")
10 ✓        if id in self.current_services:
11            raise ValueError("You already requested this!")
12        self.current_services.add(id)
13        print(f"starting {id}")
14        await async_spam(2)
15        self.current_services.remove(id)
16        return id
17
18 service = MyCoolService()
```

```
1 ✓ async def unsafe(id):  
2     return await service.my_slow_action(id)  
3  
4 ✓ try:  
5     await asyncio.gather(*[unsafe(i) for i in range(20)])  
6 ✓ except RuntimeError as e:  
7     print(repr(e))  
8     print("told you!")  
9  
10 service.current_services.clear()
```

```
starting id=0  
starting id=1  
starting id=2  
starting id=3  
starting id=4  
starting id=5  
starting id=6  
starting id=7  
starting id=8  
starting id=9  
starting id=10  
RuntimeError('You broke my service :(')  
told you!
```

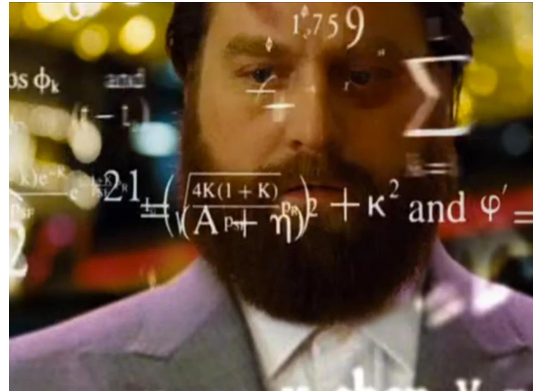
```
1 semaphore = asyncio.Semaphore(8) # less than what the service handles :)
2
3 v async def safe(id):
4 v     async with semaphore:
5         return await service.my_slow_action(id)
6
7 await asyncio.gather(*[safe(i) for i in range(20)])
```

```
starting id=0
starting id=1
starting id=2
starting id=3
starting id=4
starting id=5
starting id=6
starting id=7
starting id=8
starting id=9
starting id=10
starting id=11
starting id=12
starting id=13
starting id=14
starting id=15
starting id=16
starting id=17
starting id=18
starting id=19
```

Other synchronization primitives:

- **Lock**: only one task can access a resource at a time
- **Event**: notify multiple tasks that a specific event has occurred
- **Condition**: Combines **Lock** and **Event**
- **Barrier**: Makes all tasks wait at a specific point (like a checkpoint)

# Key Takeaways and Conclusion



“Faster execution ~~because~~ **if** your program doesn’t stay idle while waiting for a result”

- Asynchronous programming is more than just adding `async/await`! **You** make the asynchrony happen, not the keywords.
- Easy wins: **`async def`** only for coroutines, asynchronous libraries when in coroutines.
- Having `await` everywhere is a code smell: Be mindful of *why* you’re awaiting.
- Use the tools: **`gather`** and **`TaskGroup`** FTW
- Be curious! Stay on the lookout for improvements and better ways of doing things!

# Thank you!



Slides, code and references:



[jpvanegasc/async-python-talk](https://github.com/jpvanegasc/async-python-talk)