Python Async Patterns Cheat Sheet

5 Production-Tested Concurrency Patterns

Based on 72-hour real workload benchmark · Jackson Studio

Pattern	Throughput	Best For
Queue (■)	**2,847 req/sec**	**Batch API calls**
Semaphore	2,491 req/sec	Simple rate limiting
Backpressure	~2,300 req/sec	Streaming pipelines
Circuit Breaker	N/A (reliability)	Flaky external APIs
Adaptive	Self-tunes	Unknown load patterns

Pattern 1: Queue-Based Concurrency (■ Fastest)

2,847 req/sec — beat semaphore by 356 req/sec in 72-hour benchmark

```
import asyncio

async def queue_worker(queue: asyncio.Queue, results: list):
    while True:
        task = await queue.get()
        if task is None: break
        try:
            result = await process_task(task)
            results.append(result)
        except Exception as e:
            results.append({"error": str(e), "task": task})
        finally:
            queue.task_done()

async def run_with_queue(tasks: list, workers: int = 10):
    queue = asyncio.Queue()
    results = []
    worker_tasks = [
        asyncio.create_task(queue_worker(queue, results))
        for _ in range(workers)
    ]
    for task in tasks:
        await queue.put(task)
    for _ in range(workers):
        await queue.put(None)  # Poison pill to stop workers
    await asyncio.gather(*worker_tasks)
    return results

results = asyncio.run(run_with_queue(my_tasks, workers=20))
```

Pattern 2: Semaphore-Based Rate Limiting

Simple, predictable — perfect when you know your concurrency ceiling

```
import asyncio, aiohttp

async def fetch_with_semaphore(session, url, semaphore):
    async with semaphore:
        try:
            async with session.get(url, timeout=aiohttp.ClientTimeout(total=10)) as resp:
                return {"url": url, "status": resp.status, "data": await resp.json()}
        except asyncio.TimeoutError:
            return {"url": url, "error": "timeout"}

async def batch_fetch(urls: list[str], concurrency: int = 50):
    semaphore = asyncio.Semaphore(concurrency)
    async with aiohttp.ClientSession() as session:
        tasks = [fetch_with_semaphore(session, url, semaphore) for url in urls]
        return await asyncio.gather(*tasks, return_exceptions=True)

results = asyncio.run(batch_fetch(url_list, concurrency=30))
```

Pattern 3: Backpressure Queue (No OOM Crashes)

Graceful drops instead of crashes — essential for streaming pipelines

```
import asyncio, time

class BackpressureQueue:
    def __init__(self, max_size: int = 100):
        self.queue = asyncio.Queue(maxsize=max_size)
        self.stats = {"produced": 0, "consumed": 0, "dropped": 0}

    async def produce(self, item, timeout: float = 1.0):
        try:
            await asyncio.wait_for(self.queue.put(item), timeout=timeout)
            self.stats["produced"] += 1
        except asyncio.TimeoutError:
            self.stats["dropped"] += 1  # Graceful drop

    async def consume(self):
        item = await self.queue.get()
        self.stats["consumed"] += 1
        self.queue.task_done()
        return item
```

Pattern 4: Circuit Breaker for External APIs

Auto-recovery from failures — dropped cascade error rate 94%

```
import time
from enum import Enum

class CircuitState(Enum):
    CLOSED = "closed"       # Normal — pass through
    OPEN = "open"           # Failing — reject calls
    HALF_OPEN = "half_open" # Testing recovery

class AsyncCircuitBreaker:
    def __init__(self, failure_threshold=5, recovery_timeout=30):
        self.failure_threshold = failure_threshold
```

```
        self.recovery_timeout = recovery_timeout
        self.failures = 0
        self.last_failure_time = None
        self.state = CircuitState.CLOSED

    async def call(self, coro):
        if self.state == CircuitState.OPEN:
            if time.time() - self.last_failure_time > self.recovery_timeout:
                self.state = CircuitState.HALF_OPEN
            else:
                raise Exception("Circuit OPEN — skipping call")
        try:
            result = await coro
            self.failures = 0
            self.state = CircuitState.CLOSED
            return result
        except Exception as e:
            self.failures += 1
            self.last_failure_time = time.time()
            if self.failures >= self.failure_threshold:
                self.state = CircuitState.OPEN
            raise e

breaker = AsyncCircuitBreaker(failure_threshold=3, recovery_timeout=60)
result = await breaker.call(fetch_from_api(url))
```

Pattern 5: gather() Error Handling (The One That Matters Most)

Single line change — prevents one error from killing ALL concurrent tasks

```
# ■ WRONG — one exception kills everything
results = await asyncio.gather(*[fetch(url) for url in urls])

# ■ RIGHT — errors stay isolated
results = await asyncio.gather(
    *[fetch(url) for url in urls],
    return_exceptions=True  # ← This one arg changes everything
)

errors = [r for r in results if isinstance(r, Exception)]
good_results = [r for r in results if not isinstance(r, Exception)]

print(f"Success: {len(good_results)}, Errors: {len(errors)}")
# Now handle errors separately — no cascade failure
```

Want to go deeper?

Battle-Tested Python: Production Patterns That Scale — jacksonlee71.gumroad.com/l/battle-tested-python