🧠 ASYNC MASTER CHEAT SHEET

(The “What do I do now?” + “How do I design this?” guide)

1️⃣ The Mental Model — The WHY

🔹 Sync vs Async

- **Sync** → ek kaam, uska wait, phir next
  👉 *One by one, time waste*
- **Async** → kaam start karo, wait ke time dusra kaam
  👉 *Waiting overlap, time bachta hai*

Golden line (LOCK THIS):

> Async speed CPU se nahi aati, **waiting overlap** se aati hai.

Async ka fayda **sirf I/O-bound kaamon** mein hota hai:

- network
- APIs
- database
- disk

❌ CPU-heavy loops → async se fast nahi hote.

🔹 Blocking vs Non-Blocking

- **Blocking** → poora program ruk jaata hai
  - `time.sleep()` ❌
- **Non-Blocking** → sirf current task rukta hai
  - `await asyncio.sleep()` ✅

Rule:

> Async code mein **blocking call = crime** 🚨
> Ek blocking call poore event loop ko freeze kar deta hai.

🔹 Event Loop (Traffic Police 👮)

- Event loop ek **manager / scheduler** hai
- Decide karta hai:
  - kaunsa task **RUNNING**
  - kaunsa **WAITING**
  - kaunsa **DONE**
  - kaunsa **CANCELLED**

Important truth (VERY IMPORTANT):

> Event loop ko control **sirf `await` pe milta hai**

Agar `await` nahi:

- ❌ no task switching
- ❌ starvation (ek task CPU pakad ke baith jaata hai)
- ❌ cancellation fail hoti hai

2️⃣ The Architect's Structure — The HOW

> **Async program likhne se pehle hamesha design socho, code nahi.**

Async design hamesha **2 phases** mein hota hai:

🔧 Step 1: Task define karna (`async def`)

```
async def fetch(url):
    ...
```

- `async def` function **run nahi hota**
- Ye sirf **coroutine object** banata hai

- Coroutine = pause/resume function

Rule:

> Call ≠ Run (in async)

▶️ Step 2: Task run karna (`await` vs `asyncio.run`)

`await`

```
await fetch(url)
```

- Coroutine ko **execute** karta hai
- Pause point deta hai
- Sirf `async def` ke andar allowed

`asyncio.run()`

```
asyncio.run(main())
```

- Event loop **start** karta hai
- Async program ka **entry point**

Rule:

> `await` = pause + run
> `asyncio.run` = engine start

⚡ Step 3: Multiple tasks saath mein chalana (`asyncio.gather`)

```
tasks = [fetch(url) for url in urls]
results = await asyncio.gather(*tasks)
```

- **Task Creation Phase** → loop, NO `await`
- **Execution Phase** → ONE `await gather`
- Real concurrency yahin hoti hai

❌ Galat (serial async):

```
for url in urls:
    await fetch(url)
```

Golden rule (never forget):

> Loop + await = SERIAL
> Tasks + gather = CONCURRENT

🛑 CRITICAL WARNING (VERY IMPORTANT)

Default behavior of `asyncio.gather()` is DANGEROUS in production.

❌ By default:

- Agar **ek task fail** ho gaya (e.g., ek URL crash),
- To **poora gather fail** ho jaata hai
- Baaki tasks bhi stop / cancel ho jaate hain
- Program crash ho sakta hai

```
results = await asyncio.gather(*tasks)
```

✅ The FIX (Production-safe)

```
results = await asyncio.gather(
    *tasks,
    return_exceptions=True
)
```

👉 Ab:

- Ek task fail ho sakta hai
- Baaki tasks continue karte rahenge
- Errors result list mein aa jaayenge

Rule:

> Scraping / APIs mein **hamesha** `return_exceptions=True` **use karo**

💤 Step 4: Pause without blocking (`asyncio.sleep`)

```
await asyncio.sleep(2)
```

- Sirf current task rukta hai
- Event loop dusre tasks chala sakta hai
- CPU idle / free rehta hai

❌ Never use in async code:

```
time.sleep(2)
```

3️⃣ System Design Patterns — HOW TO THINK

🧩 Pattern 1: Async Scraping / URL Fetching

Problem: Multiple URLs ka wait

Design:

1. URLs list
2. Async fetch function (one URL)
3. Tasks create karo
4. `gather` se execute

```
tasks = [fetch(url) for url in urls]
results = await asyncio.gather(*tasks, return_exceptions=True)
```

🧩 Pattern 2: API Fan-Out (Backend Design)

Problem: Ek request → multiple APIs

❌ Galat:

```
a = await api1()
b = await api2()
c = await api3()
```

✅ Sahi:

```
tasks = [api1(), api2(), api3()]
a, b, c = await asyncio.gather(*tasks, return_exceptions=True)
```

Concept:

> Fan-out (spread calls) → Fan-in (collect results)

🧩 Pattern 3: Rate-Limited Async (Avoid Ban / 429)

Problem: Too many requests at once

Tool: `asyncio.Semaphore`

```
sem = asyncio.Semaphore(5)

async def safe_fetch(url):
    async with sem:
        return await fetch(url)
```

Then:

```
tasks = [safe_fetch(url) for url in urls]
await asyncio.gather(*tasks)
```

Rule:

> Async ≠ unlimited speed
> Async = controlled concurrency

🧩 Pattern 4: Background Tasks (Use Carefully)

```
task = asyncio.create_task(job())
```

Use for:

- logging
- cleanup
- heartbeats

⚠️ Danger:

- Orphan tasks
- Memory leaks
- Long-running servers crash

Rule:

> `create_task()` sirf tab use karo jab tum task ko track / cancel kar sakte ho.

4️⃣ The Decision Flow — Rules of Thumb

🧭 Blank Screen Decision Guide

- **Multiple URLs / APIs ka wait?**
  👉 `asyncio.gather()`
- **Delay chahiye async code mein?**
  👉 `await asyncio.sleep()`
- **Too many requests / ban ka risk?**
  👉 `asyncio.Semaphore()`
- **Background kaam (fire-and-forget)?**
  👉 `asyncio.create_task()` (with care)

- **Loop ke andar `await` likhne ka mann ho raha hai?**
  👉 ❌ STOP — design galat hai
- **Task cancel nahi ho raha?**
  👉 Check: kya task `await` pe ja raha hai?
- **Program 2 din baad crash?**
  👉 Orphan tasks / memory leak suspect karo

🔑 Key Technical Terms (All-in-One)

- **Coroutine** → pause/resume function
- **Task** → coroutine under event loop control
- **Event Loop** → async task scheduler
- **RUNNING / WAITING / DONE / CANCELLED** → task lifecycle
- **Blocking Call** → freezes loop
- **Non-Blocking Call** → allows switching
- **Concurrency** → ek time pe multiple waiting tasks
- **Parallelism** → multiple CPU cores (NOT async)
- **Semaphore** → concurrency limiter
- **Fan-out / Fan-in** → async API pattern
- **Orphan Task** → background task with no owner
- **Starvation** → event loop ko control hi na milna

🔒 Final One-Line Mental Model (Never Forget)

> **Async ka matlab hai:**
> “Saare kaam pehle event loop ko de do,
> phir control chhod do —
> aur failure ko bhi design karo.”

🧠 FINAL NOTE (VERY IMPORTANT)

If you ever feel stuck, don’t ask:

> “Code kaise likhun?”

Ask:

> **“Event loop ke paas kitne tasks honge?”**

If answer = **1** → ❌ slow
If answer = **many** → ✅ async working