3D Printing Queue Simulator – Technical Exercise

1 — Objective

Build a **simulator** that manages 3D printers and a queue of printing jobs.
The candidate should deliver a working solution (CLI or small service) with clean code, basic automated tests, and a README explaining how to run the program.

---

2 — Requirements

MUST (minimum deliverable)

1. **Job model** with fields: id, material, est_time (seconds), priority (integer), created_at.
2. A queue structure that processes **jobs by priority** (higher priority first — define and document whether a lower or higher number is "more urgent").
3. Simulate processing of the jobs by **N printers** (configurable).
4. **Handle concurrency/state safely** — for example using locks, asyncio, or a thread-safe priority queue.
5. Output a final report (JSON or CSV) with the status of each job: queued, running, completed, cancelled and timestamps: started_at, finished_at.
6. Provide instructions in the README and usage examples.
7. Basic unit tests (pytest) covering enqueueing, job ordering and final report.

SHOULD

1. Simple CLI to: add jobs, list queue, cancel jobs.
2. Metrics report: average waiting time, throughput, total printer occupancy time.
3. Simulation with a time scaling factor (for example time_scale) so you do not have to wait real seconds.

OPTIONAL / BONUS

1. Dynamic priority (increase priority of jobs waiting too long).
2. Preemption: a higher-priority job interrupts a running job.
3. Simple persistence (SQLite) for job history.
4. REST API (FastAPI) to create/list/cancel jobs.
5. Visualization (e.g. with matplotlib) of printer utilization or a Gantt chart.

---

3 — Data Model (example JSON for a job)

{

"id": "job-0001",

```
  "material": "PLA",
  "est_time": 120.0,
  "priority": 1,
  "created_at": 1690000000.0
}
```

---

4 — Scheduling Rules

- **Priority policy:** Jobs are processed by priority; between jobs with the same priority, follow FIFO (first-in, first-out).
- If there are M printers available, the M highest-priority jobs are assigned simultaneously.
- Simulation of time: use a **time scaling factor** (time_scale) to shorten the actual waiting time (for example, est_time * time_scale).
- Cancelling jobs:
  - If job is queued → remove it.
  - If job is running → optional; may mark it as cancelled and free the printer.
- Final report must include for each job:
  - wait_time = started_at - created_at
  - run_time = finished_at - started_at.

---

5 — Suggested Architecture (example)

- models.py — Job, Printer (dataclasses).
- queue_manager.py — priority queue implementation.
- simulator.py — logic of workers (one coroutine or thread per printer).
- cli.py — command line interface for adding/cancelling/listing.
- tests/ — pytest test cases.
- README.md — instructions and how to run.

This structure is a suggestion; candidates may choose a different layout.

---

6 — Implementation Justification (open answer)

Candidate field:
Explain and justify your chosen approach to concurrency, your queue implementation, and how you ensured thread-safety or correct synchronization of the printer workers.

---

7 — Important Tips

- **Avoid long real sleeps —** always expose a time_scale factor for testing.
- Use dataclasses with field(compare=False) for objects inside a priority queue.
- For stable ordering, include an incremental counter alongside priority to avoid comparing Job objects directly.
- Provide clear logging with ISO timestamps to help reviewers follow the simulation.

---

8 — Short Example: Structs + Worker (Python pseudocode)

```
from dataclasses import dataclass, field

import time

import asyncio

from typing import Optional


@dataclass(order=True)

class PrioritizedItem:

    priority: int

    counter: int
```

```python
    job: "Job" = field(compare=False)


@dataclass
class Job:
    id: str
    material: str
    est_time: float
    priority: int = 0
    created_at: float = field(default_factory=time.time)
    started_at: Optional[float] = None
    finished_at: Optional[float] = None
    status: str = "queued"


async def printer_worker(printer_id: int, queue: asyncio.PriorityQueue,
                         state: dict, time_scale: float):
    while True:
        priority, counter, job = await queue.get()
        job.status = "running"
        job.started_at = time.time()
        # simulate printing with a time scale
        await asyncio.sleep(job.est_time * time_scale)
        job.finished_at = time.time()
        job.status = "completed"
        queue.task_done()
```

Note: invert the meaning of priority if you prefer “higher number = more urgent”, but document your decision.

---

9 — Example Scenarios for Testing

Scenario A — Single printer, order by priority/FIFO

Jobs (arrival time = 0):

- J1: priority=1, est_time=2s

- J2: priority=2, est_time=1s
- J3: priority=1, est_time=3s

If “lower number = higher priority” and with one printer:

- Execution order: J1 (priority 1, arrived before J3) → J3 → J2.

With time_scale = 0.1, run times become 0.2s, 0.3s, 0.1s.

Scenario B — Two printers (timeline)

Jobs (t=0):

- J1: priority=1, est_time=10s
- J2: priority=2, est_time=5s
- J3: priority=1, est_time=3s

With two printers:

- t=0: Printer1 → J1 (10s), Printer2 → J3 (3s)
- t=3s: Printer2 free → J2 (5s) → finishes at t=8s
- J1 finishes at t=10s.

---

10 — Pytest Example Skeleton

```
def test_priority_order():
  sim = Simulator(num_printers=1, time_scale=0.01)
  sim.add_job(Job(id="J1", est_time=2, priority=1))
  sim.add_job(Job(id="J2", est_time=1, priority=2))
  sim.add_job(Job(id="J3", est_time=3, priority=1))
  sim.run_until_complete()
  report = sim.get_report()
  order = [j['id'] for j in report if j['status'] == "completed"]
  assert order == ["J1", "J3", "J2"]
```

---

11 — Metrics and Final Report

For each job:

- id
- priority
- created_at

- started_at
- finished_at
- status
- wait_time
- run_time

Global summary:

- avg_wait_time
- median_wait_time
- throughput = total jobs / total simulation time
- Printer utilization = busy_time / total simulation time (per printer)

Report format: a single JSON or CSV file, e.g. report.json.

---

12 — Suggested Time Allocation (3–4 hours)

- 15–30 min: read specification, plan data model and APIs.
- 90–120 min: implement queue, workers, enqueuing and basic report.
- 30–45 min: write tests and adjust time_scale.
- 20–30 min: write README, polish and provide examples.
- 15–30 min: (optional) extra features such as cancellation or metrics.

---

13 — Deliverables

- Git repository containing:
    - README.md with instructions, explanation of time_scale, and examples.
    - requirements.txt.
    - Source code (e.g. simulator/ folder).
    - tests/ with pytest cases.
    - sample_input.json and an example sample_report.json.
- Minimum execution command, for example:
- python cli.py --input sample_jobs.json --printers 2 --time-scale 0.01
- Tests execution:
- pytest -q