1) measuring a process — what to log and why

think of a process as a conveyor belt for “units of work” (an order, a student, an invoice). you want to *see* the belt in time. that means **timestamps** and a few derived measures.

a) the minimum timestamps (write these on your diagrams)

for each unit (e.g., “order #123”):

- **queued_at** – when the customer starts waiting for a step
- **start_at** – when work on that step actually begins
- **finish_at** – when work on that step ends
- **handoff_at** – when it moves to the next step (sometimes same as finish_at)

for Starbucks, log for each step: `order`, `pay`, `barista_start`, `barista_finish`, `pickup`.

b) the derived measures (what you calculate from timestamps)

- **wait time** at a step = `start_at − queued_at`
  plain words: “time i was idle in line before anyone touched my work.”
- **service time** (a.k.a. **cycle time for the step**) = `finish_at − start_at`
  “hands-on time.”
- **lead time** (end-to-end for the unit) = `final_finish − first_queued`
  “door-to-door time the customer experiences.”
- **throughput** = units completed per hour (or per minute)
  “how fast the belt spits out finished units.”

- **WIP (work in process)** = how many units are currently in system
  "how crowded the belt is right now."

one super-useful relationship (Little's Law):
WIP = Throughput × Lead time.
if lead time is long at the same throughput, your WIP must be big (things are stuck). in practice, when you reduce wait/cycle at a bottleneck, lead time falls and WIP follows.

c) value-added vs non-value-added (VA vs NVA)

- **VA**: if the customer would pay for that minute (brewing the drink).
- **NVA**: they wouldn't pay (standing in line, typing data twice, hunting a file).
  goal of BPR: **squeeze NVA** without hurting VA quality.

---

2) spotting bottlenecks — the simple way (no formulas first)

rule of thumb:

> **the bottleneck is the step with the longest average service time *or* the longest queue, and therefore the smallest effective capacity.**

how to see it quickly:

1. draw the steps left→right.
2. write the average **service time** under each step (from a quick time study: take 10 samples per step).
3. stand in the store (or imagine it): where do people pile up? that's your queue; bottleneck is usually just downstream.

Starbucks, **As-Is**:

- cashier: 20–25s per order (ask, customize, key in, take cash/card)
- barista: 40–60s (depends on drink)
- pickup: negligible service time; often *wait* because barista is behind

bottleneck? **barista** (longest hands-on). but the **cashier** can *starve or flood* the barista. if cashier slows, barista idles; if cashier speeds up, the pickup area overflows. reengineering that removes cashier keystrokes (app pay) stabilizes the feed so the barista gets evenly paced, earlier jobs.

deeper view: utilization

- **capacity** of a step ≈ `3600 / avg_service_time_in_seconds` (units/hour, single worker)
- **utilization** ≈ `arrival_rate / capacity`. when utilization > ~80–85%, queues explode (variability effect).
  so you aim to keep your slowest step (the bottleneck) **below ~80% utilization** by either:
- lowering service time (automation, better layout, pre-batching), or
- adding parallel servers (2 baristas), or
- smoothing arrivals (app ETA staggering), or
- moving work *away* from that step (do milk-foaming in parallel, pre-grind, etc.)

---

3) spreadsheet simulation — the "evidence" sheet you can build in 10 minutes

make a sheet with **one row per order** (or per student, invoice, etc.). you only need 10–30 rows to reason about cause→effect.

columns to create (copy this into your notebook)

A) `order_id` (1..N)
B) `arrival_time` (in seconds from opening; or just 0,30,60,… for every 30s)
C) `service_cashier` (sec)
D) `service_barista` (sec)
E) `start_cashier` = `MAX(arrival_time, finish_cashier_of_prev)`
F) `finish_cashier` = `start_cashier + service_cashier`
G) `start_barista` = `MAX(finish_cashier, finish_barista_of_prev)`
H) `finish_barista` = `start_barista + service_barista`
I) `lead_time` = `finish_barista − arrival_time`
J) `wait_cashier` = `start_cashier − arrival_time`
K) `wait_barista` = `start_barista − finish_cashier`

explain what's happening:

- each order **can't** start at cashier before it arrives; it also can't start before the cashier finishes the previous customer → that's the `MAX`.
- the barista can't start until (a) cashier finished sending the spec **and** (b) barista is free. again `MAX`.
- this already models a two-station queue with realistic blocking.

now create To-Be scenarios

- **App pay**: set `service_cashier = 0` (or 2–3s if you want a quick ID check).
- **Parallel barista**: create `barista_A` and `barista_B` columns and assign every other job to A/B (or "shortest queue first").
- **ETA staggering**: increase the spacing in `arrival_time` for app orders to reduce clumps.

what to compare after filling 20–30 rows

- average `lead_time` (customer-experienced time)
- max `lead_time` (worst-case)
- average `wait_barista` (how badly you starved/flooded the station)

- #orders finished per 10 minutes → throughput
- from these, you can argue "our To-Be reduces lead time by X% and raises throughput by Y%."

---

4) merit function — turning many goals into one score

why we need it

in an exam (or real life), you'll have multiple knobs: **cost per transaction**, **lead time**, **errors**, **automation ratio**, **CSAT**, etc. a **merit function** lets you write one number that captures the trade-off. then you can say: "we choose the design with **lower M** (or higher, depending how we set it)."

step-by-step build (with explanation)

step 1: pick the criteria (keep it 3–6 items you can defend)

- `COST_TXN` — total cost per order (wages + payment fees + shrinkage…)
- `LEAD` — average end-to-end minutes (customer experience)
- `THROUGHPUT` — orders/hour (capacity to grow revenue)
- `ERRORS` — remakes/100 orders (quality, waste)
- `AUTOMATION` — % of steps done by system (maintainability, scalability)

step 2: make them unit-free by normalizing
we can't add minutes to rupees to percentages. so convert each to **0..1** scale:

- for "lower is better" (cost, lead, errors):
  `norm = (value − best) / (worst − best)` → 0 means best, 1 means worst
- for "higher is better" (throughput, automation):
  `norm = (best − value) / (best − worst)` → still 0 means best, 1 worst

this way, all criteria are “bad when large”. easy to sum with weights.

step 3: assign weights `w_i` that sum to 1

- **COO focus** → heavier on `LEAD` and `THROUGHPUT`
- **CFO focus** → heavier on `COST_TXN`
- **CTO focus** → some weight on `AUTOMATION` (for scalability, reliability)
  typical exam-safe set:
  `w_cost=0.35, w_lead=0.25, w_thru=0.2, w_err=0.15, w_auto=0.05`

step 4: compute merit
`MERIT = Σ (w_i × norm_i)` → **lower is better**.

step 5: sanity check

- if two designs tie on merit, present the **spider chart** idea verbally: “Design A cheaper but slower; Design B faster but slightly costlier. If traffic is high, we prefer B.” (this shows you *understand* the trade-off, not just the math.)

tiny numeric illustration (so the logic sticks)

Assume As-Is vs To-Be:

- `COST_TXN` (rupees): **As-Is 130**, To-Be 118
- `LEAD` (min): As-Is **12.0**, **To-Be 7.0**
- `THROUGHPUT` (orders/hr): As-Is **120**, **To-Be 170**
- `ERRORS` (per 100): As-Is **3.5**, **To-Be 2.0**
- `AUTOMATION` (% steps auto): As-Is **15%**, **To-Be 65%**

set “best/worst” as the min/max among designs (or include a target). normalize each, multiply by weights, sum → you’ll see To-Be’s MERIT clearly lower. you can actually do this on paper in under 2 minutes.

5) proving improvement — ROI, payback, sensitivity (with explanation)

even in a process exam, a small finance frame scores big.

a) ROI & payback (plain and fast)

- **incremental profit per period** = `(Δthroughput × margin) − Δopex`
- **ROI** = `incremental profit / capex`
- **payback (months)** = `capex / incremental monthly profit`

explain it in words: “we spend once (capex on app) to permanently cut cashier time (opex) and increase capacity (throughput) so more orders fit in peak hours. the extra gross margin per hour repays the app in N months.”

b) sensitivity (why this is powerful in answers)

ask “what if i’m wrong by 10–20%?” vary 1–2 inputs:

- if adoption is lower (only 40% use the app), do we still win?
- if card fee rises by 0.3%, does kiosk make more/less sense than mobile?
- if drink mix shifts to more complex items (barista time +10s), do we need a second barista?

you don’t need charts—just say: “even with 20% fewer app users, payback extends from 6 to 8 months but remains acceptable.” that shows **robustness**.

---

extra tools you can drop in any case

- **takt time** (pace of customer demand): `available_time / demand`. if barista cycle time > takt time, demand outpaces capacity → queue grows. solution: reduce cycle time or add capacity.

- **utilization target**: aim for ≤80–85% at bottleneck to avoid runaway queues when variability hits.
- **first-time-right**: improve spec clarity so remakes drop. a single remake can add 60–90s → it's secretly huge on lead time.
- **layout & movement**: shave seconds by reducing walking/reach (put milk, cups, syrups within 1–2 steps). cumulative seconds matter at scale.
- **pre-work**: move prep tasks before the bottleneck (e.g., pre-label cups when order arrives).

---

how this maps to your two anchor cases

Starbucks

- measurement = timestamps for order/pay/start/finish/pickup
- bottleneck = barista; app reduces cashier time, smooths arrivals, pushes clean specs
- spreadsheet = two stations with `MAX()` logic; scenario: app pay (cashier ~0), ETA staggering, add 2nd barista at peaks
- merit = weights on cost/lead/throughput/errors/automation → To-Be wins
- ROI = extra orders/hour × margin – opex change, divided by app capex → payback in months

University fees / enrollment

- units = student-course selections and fee status updates
- bottlenecks = manual verification and exception handling
- reengineering = *rules first* (eligibility by completed pre-reqs + due fees must be "OK") so bad choices never enter the system

- spreadsheet = treat each “approval step” like a station; remove the ones the rules eliminated
- merit = error rate weight higher (registrar cares), lead time for student schedule, staff-hour cost; To-Be scores lower merit by eliminating exceptions

---

a tiny “exam paragraph” you can adapt (keep this wording vibe)

> “We measured the process using event timestamps per order (queued, start, finish per station). The longest average service time and queue formed at the barista, making it the bottleneck. By shifting payment and specification capture to the mobile app, we eliminated front-counter keystrokes, smoothed arrivals, and fed complete specs directly to the barista screen. A 20–30s reduction at the front and a 10–15s barista improvement from fewer remakes reduced average lead time from ~12 minutes to ~7 minutes and raised throughput from ~120 to ~170 orders/hour. A weighted merit function (Cost 0.35, Lead 0.25, Throughput 0.20, Errors 0.15, Automation 0.05) favored the To-Be design across all tested scenarios. With conservative adoption, incremental gross margin covers the app capex within 6–8 months, remaining positive under ±20% sensitivity.”

---

if this framing works, next i can give you:

- a **fill-in-the-blanks merit function sheet** (you can copy by hand),
- a **ready-made timestamp table template** (so you just plug numbers), and
- **exam-ready short/long answers** for: takt time, WIP/throughput/lead, Little’s Law, bottleneck, utilization, VA vs NVA, and change-management lines to close any case.