**Day 23-25**

**What we’ve done so far (context mapped to the pseudocode)**

Brief, precise mapping of project progress (so you can see how the pseudocode fits with actual work done):

* **Day 15** — *User request generation*: random requests with {cpu, memory, storage, deadline, request\_time}. (This produces the raw list of demands.)
* **Day 16** — *Fog nodes with limited resources*: FogNode class tracking available\_cpu, available\_mem, available\_storage (this provides capacities C\_k in pseudocode).
* **Day 17** — *Baseline random assignment*: we implemented random assignment fog\_id for each user (this corresponds to l ← rand(list) and initial random ordering).
* **Day 18** — *Fixed pricing*: base\_price = cpu\*PRICE\_CPU + memory\*PRICE\_MEM (used when rounds ≤ W as \hat P\_i).
* **Day 19–21** — *Test & visualize baseline* (cost vs demand scatter + line).
* **Day 22** — *Utility functions*: user\_utility(cost, latency) and provider\_utility(price, resource\_cost).
* **Day 23** — *Dynamic pricing rule*: add dynamic\_price(base, demand, capacity, gamma); this is a simple instance of the price-update step in Algorithm 2.
* **Day 24** — *Iteration loop*: implemented rounds where users choose fog → fog updates price → repeat (our prototype loop).
* **Day 25 (upcoming)** — *Stopping condition (equilibrium)*: implement termination when assignments / prices stabilize (we will annotate and implement below).

Now I’ll comment **each line / block** of the uploaded pseudocode (Algorithm 1 and Algorithm 2) exactly as written, and show how each maps to concrete code/data structures and to things we’ve already implemented.

**Annotated pseudocode — Algorithm 1: Main**

(Original pseudocode lines reproduced, with line-by-line commentary and implementation notes.)

1: begin

* Start of the main routine. In code this is the main() function or top-level simulation driver.

2: Let Tr be current time.

* Tr = current observation time (timestamp).
* Implementation: Tr = int(time.time()) or logical time in the simulation. We used RequestTime earlier; Tr is the time when you start collecting sample demands.

3: Let x represent the time from when the collection of demand starts.

* x = lookback window length (e.g., 30 minutes or 30 seconds in simulation).
* Implementation: define x\_seconds = 30 or a parameter LOOKBACK\_WINDOW. This determines which historical lists list\_z you will aggregate.

4: for ∀z ∈ {Tr − x, Tr} do

5: list ← list ∪ listz

6: end for

* For every time-slot z between Tr-x and Tr, collect all demands that arrived in that slot and aggregate into list.
* Implementation details:
  + list\_z is the set of demands logged at time z (or in a sub-window). Practically you will sample multiple z windows (e.g., partition Tr-x..Tr into smaller windows or sample random z values).
  + Code: collected\_list = [] then for each z append demands\_by\_time[z].
* Purpose: build the pool of sample demands from which L\* and L\*\* will be derived later.

7: l ← rand(list) /\* random list \*/

* Shuffle list randomly to get l (random processing order). Everyone has equal chance to be processed first.
* Implementation: random.shuffle(list) or l = random.sample(list, k=len(list)).

8: for i = 1 to |list| do

9: boolean = true

* Iterate over each demand in the randomized list l. Initialize a flag boolean (we’ll use it to check capacity feasibility for this demand over its requested interval).

10: for k = 1 to |listi(Di)| do

11: if listi(Di · ˆ Ck) ≤ Ck ∀t ∈ (Tr, t∗) then

12: Ck ← (Ck − listi(Di · ˆ Ck))

13: else

14: boolean = false

15: end if

16: end for

* This inner loop checks **resource-wise feasibility** for the ith request across all resource components k (e.g., k indexes CPU, RAM, Storage).
  + listi(Di) denotes the demand vector for the i-th entry: Di = sum of requested components (or vector per resource). |listi(Di)| = number of resource components.
  + Ĉ\_k (hat Ck) represents the unit capacity multipliers or per-component demand scaling (in your earlier notation Di·Ĉ\_k yields actual capacity needed for resource k).
  + Condition listi(Di · Ĉ\_k) ≤ C\_k ∀ t ∈ (Tr, t\*) means: for the entire interval from now (Tr) to the user’s deadline t\* (the window in which the resource is reserved), the **required amount** for resource k must be ≤ currently available capacity C\_k. This enforces time-windowed resource feasibility.
* If feasible for all components k, the code **reserves** capacity by subtracting listi(Di · Ĉ\_k) from C\_k.
* If any component fails, set boolean = false (we do not change capacities permanently).
* Implementation mapping:
  + for k, resource\_name in enumerate(resource\_types):
  + required = demand\_vector[k] \* resource\_unit\_scale[k] (this is listi(Di · Ĉ\_k))
  + Check if required <= available\_C[k]:
    - Option: implement temporary reservation then commit after loop, or commit as you go and reset() on failure (below).
  + The ∀ t ∈ (Tr, t\*) aspect: if you support time-varying reservations you must track capacity per time-slot, not only an instantaneous number. A simple implementation assumes immediate allocation and decrement until t\*.

17: if boolean = false then

18: reset()

19: end if

* If feasibility failed, call reset() which reverts any tentative allocations we made during the current i check. This ensures capacities are unchanged if the request is rejected.
* Implementation detail:
  + Use **transactional** approach: before the inner loop, copy capacities temp\_C = C.copy(), apply tentative temp\_C -= required. If all pass, commit C = temp\_C. If not pass, do nothing. That avoids an explicit reset() function. If you use in-place updates, implement reset() to restore capacities from a backup saved before checking i.

20: Price()

* Call the Price() subroutine (Algorithm 2) to compute the price P\_i for the current agent (based on historical data and the t\* window).
* Implementation mapping:
  + P\_i = Price(current\_agent, collected\_list, C, parameters) — ensure Price() uses the same Tr and t\* context.

21: end for

22: return

23: end

* Finish processing all demands and return.

**Annotated pseudocode — Algorithm 2: Price Calculation**

(Again, commented line-by-line, faithful to the uploaded text.)

1: The price is first established using the formula ΣΣ Di·Ci depending on the current demand for several rounds.

* Initialization: compute a base price using demand × cost/capacity metric. In practice you can define:
  + base\_price = sum\_over\_components( Di\_component \* cost\_per\_unit\_component )
  + Or the notation ΣΣ Di·Ci suggests summing Di \* C\_i across components i and possibly across a few rounds to smooth.
* This is the initial price used for early rounds before you have enough history.

2: if no. of rounds ≤ W then ▷ /\* W is set by the system \*/

3: Pi ← \hat Pi ▷ /\* \hat Pi is system generated threshold price \*/

* **Warm-up stage**: for the first W rounds (bootstrap), do not compute the full dynamic price — instead use a system threshold price \hat P\_i (a stable base price). This avoids volatility early on.
* Implementation:
  + Maintain round\_count per agent or a global round counter. If round\_count <= W: P\_i = P\_hat\_i.
  + P\_hat\_i can be computed as base price or an initial heuristic (example: P\_hat\_i = base\_price \* (1 + small\_margin)).

4: else

5: s ← rand(Tr, t∗i)

6: L\* ← process(s)

7: L\*\* ← process(s)

* Otherwise (after warm-up rounds) pick random sample windows s in the interval (Tr, t\*ᵢ) (this is the sample selection step q = q1,..., q\_l\* referenced earlier).
  + s ← rand(Tr, t\*ᵢ) : pick s as one or more random time windows inside (Tr, t\*ᵢ).
* process(s) extracts the sample requests in s and splits them into:
  + L\* = set of **similar** requests (to the current agent) — similarity based on demand-vector closeness, weighted by resource cost preferences (e.g., CPU more important).
  + L\*\* = set of **dissimilar** requests (the rest).
* Implementation:
  + Implement process(s) to:
    - Retrieve sample instances samples = get\_samples\_in\_window(s).
    - For each sample compute similarity score sim = similarity(curr\_req, sample\_req) (e.g., weighted Euclidean distance using weights w\_cpu > w\_ram > w\_disk).
    - Choose a threshold or pick top-k closest to form L\*, remainder into L\*\*.
  + The paper suggests picking multiple s windows and aggregating L lists; implement by repeating s ← rand(...) multiple times and unioning the results.

8: Qi = (Σ\_{i=1}^{|L\*|} D\_i·P\_i / Σ\_{i=1}^{|L\*|} Σ\_{j=1}^{k} D\_i·C\_j )

+ δ ( Σ\_{i=1}^{|L\*\*|} D\_i·P\_i / Σ\_{i=1}^{|L\*\*|} Σ\_{j=1}^{k} D\_i·C\_j )

▷ /\* set price by Equation 3 \*/

* This is the central Equation (3). Detailed breakdown:
  + Numerator (similar group): ∑\_{i∈L\*} D\_i · P\_i
    - D\_i = total unit demand for sample i (sum of components).
    - P\_i = historical price at which that sample was served (from your Table1).
  + Denominator (similar group): ∑\_{i∈L\*} ∑\_{j=1}^{k} D\_i · C\_j
    - C\_j = capacity or cost factor for component j (component-level scaling).
    - The denominator is a weighted sum across resource components to normalize by capacity usage.
  + Second term (dissimilar group) is the same structure multiplied by δ (0 < δ < 1) to control how much influence dissimilar samples have.
* Implementation notes:
  + Compute the sums with care to avoid division by zero (if L\* or L\*\* empty).
  + Vectorized implementation:
  + numer\_sim = sum(D\_i \* P\_i for each sample in L\_star)
  + denom\_sim = sum(sum(D\_i \* C\_j for j in components) for sample in L\_star)
  + term\_sim = numer\_sim / denom\_sim if denom\_sim>0 else fallback
  + numer\_dissim = ...
  + denom\_dissim = ...
  + Qi = term\_sim + delta \* term\_dissim
  + Qi is the proposed price for the current agent calculated from historic samples.

9: if Pi ≤ δi then

10: final allocation Pi = Qi

11: else

12: reject

13: end if

14: end if

15: return

* **Important note about this check (ambiguity):**
  + The uploaded image line if Pi ≤ δi then is ambiguous / likely OCR or typesetting confusion. Based on the textual description earlier in the paper and standard logic, the correct condition should be:
    - if Qi ≤ γ\_i then final allocation (Pi = Qi) else reject
    - Here γ\_i is the agent’s **maximum willingness to pay** (their valuation). So you accept the user if the computed price Qi is ≤ the user’s valuation γ\_i.
  + Another possibility is comparing to a threshold \hat q\_i (system-generated threshold), but the standard acceptance test is user valuation γ\_i.
* Implementation: **use** if Qi <= gamma\_i: accept & set Pi = Qi; else reject.
* After acceptance, the fog node must commit resource capacities for the interval (Tr,t\* ) (we already reserved tentatively in Algorithm 1).

**Concrete implementation mapping (how to turn the pseudocode into code — full details)**

Below I give exact implementation steps and representative code snippets for each subcomponent called by the pseudocode. Use these to build/replace functions in your existing Python simulator.

**Data structures**

# Resource components mapping

resource\_types = ['cpu', 'ram', 'storage'] # k = 3

# available capacities per fog node (per component)

fog\_node = {

'id': 1,

'capacity': {'cpu': 16, 'ram': 64, 'storage': 500},

'available': {'cpu': 16, 'ram': 64, 'storage': 500},

'base\_price': 10.0,

'assigned\_requests': [] # list of request objects

}

# Request object (list element)

request = {

'id': 123,

'demand\_vector': {'cpu': 4, 'ram': 8, 'storage': 100},

'D\_i': 4+8+100, # optionally store sum

't\_star': Tr + 30, # deadline (absolute)

'request\_time': Tr - 5,

'gamma': 120.0, # user's max willingness to pay

'history\_price': None # when stored from past data

}

**collect\_list(Tr, x) — implementing lines 4–6**

def collect\_list(Tr, x, demand\_store):

# demand\_store maps time -> list of requests that arrived at that time

collected = []

for t in range(Tr - x, Tr + 1): # inclusive

collected.extend(demand\_store.get(t, []))

return collected

* If you prefer sampling z windows instead of iterating each second, do random sampling of z in range(Tr-x, Tr).

**Randomize list (line 7)**

import random

random.shuffle(collected) # in-place

# l = collected # in the pseudocode 'l' is that randomized list

**Capacity feasibility check (lines 8–16) — transactional approach**

def check\_and\_reserve(fog, req):

# fog.available is dict; we'll use a temp copy

temp\_avail = fog['available'].copy()

for comp in resource\_types:

required = req['demand\_vector'][comp] # Di \* Ĉ\_k if Ĉ\_k !=1

# If you manage time-slot capacities, check per time-slot instead

if required <= temp\_avail[comp]:

temp\_avail[comp] -= required

else:

return False # cannot allocate

# commit

fog['available'] = temp\_avail

fog['assigned\_requests'].append(req)

return True

* If you need to reserve for interval (Tr, t\*) per time slot, available needs to be a schedule (e.g., an array keyed by time).

**reset() behavior (lines 17–19)**

* If you updated fog.available incrementally and then found a failure, you must restore previous state. Using the transactional copy above avoids needing reset(). If you must implement reset():

def reset(fog, snapshot):

fog['available'] = snapshot

# optionally remove last tentative assignment from assigned\_requests

* Implementation advice: always work on temp and commit once success.

**Price() subroutine (Algorithm 2) — exact mapping**

def Price(request, Tr, collected\_samples, round\_count, W, P\_hat\_i, delta):

# request: current request object

if round\_count <= W:

return P\_hat\_i # simple threshold

# else:

# s <- rand(Tr, request['t\_star'])

s = pick\_random\_time\_windows(Tr, request['t\_star'])

samples = get\_samples\_in\_windows(s, collected\_samples) # union of instances

L\_star, L\_dstar = split\_similar\_dissimilar(samples, request)

Qi = compute\_Qi(L\_star, L\_dstar, resource\_costs, delta)

# acceptance test: compare Qi to gamma\_i (user valuation)

if Qi <= request['gamma']:

return Qi

else:

return None # indicate reject

Break down the helper functions:

**pick\_random\_time\_windows(Tr, t\_star)**

* Return one or multiple random windows s within (Tr, t\_star) as described in the paper (q1.. q\_l\*). Implementation example: randomly pick L anchor times and take small windows around them.

**get\_samples\_in\_windows(s, collected\_samples)**

* Use collected\_samples (from collect\_list) and filter by whether sample.request\_time falls inside any s window.

**split\_similar\_dissimilar(samples, request)**

* Compute weighted distance between request.demand\_vector and each sample’s demand\_vector.
* Use a threshold sim\_threshold or pick top-k closest\_count for L\*.
* Example distance metric (weighted Euclidean):

weights = {'cpu': 3.0, 'ram': 2.0, 'storage': 1.0} # CPU > RAM > storage

def similarity\_score(req1, req2):

s = 0.0

for comp in resource\_types:

diff = req1['demand\_vector'][comp] - req2['demand\_vector'][comp]

s += weights[comp] \* (diff \*\* 2)

return math.sqrt(s) # smaller is more similar

* Then sort samples by similarity\_score. Choose top m for L\*, rest for L\*\*.

**compute\_Qi(L\_star, L\_dstar, resource\_costs, delta)**

def compute\_Qi(L\_star, L\_dstar, resource\_costs, delta):

# resource\_costs: mapping comp -> C\_j (cost or capacity scaling)

# compute numerator and denominator for L\*

numer\_sim = sum(sample['D\_i'] \* sample['P\_i'] for sample in L\_star)

denom\_sim = sum(sum(sample['D\_i'] \* resource\_costs[comp] for comp in resource\_types)

for sample in L\_star)

term\_sim = numer\_sim / denom\_sim if denom\_sim > 0 else 0.0

numer\_dissim = sum(sample['D\_i'] \* sample['P\_i'] for sample in L\_dstar)

denom\_dissim = sum(sum(sample['D\_i'] \* resource\_costs[comp] for comp in resource\_types)

for sample in L\_dstar)

term\_dissim = numer\_dissim / denom\_dissim if denom\_dissim > 0 else 0.0

return term\_sim + delta \* term\_dissim

* Edge-case handling: if both L\* and L\*\* empty, fallback to P\_hat\_i or base\_price.

**Stopping condition / equilibrium (Day 25)**

You asked explicitly to implement the stopping condition. The pseudocode does not spell it out fully — it loops through list once — but for iteration across rounds we must define termination. Here are precise, non-simplified conditions you can implement (choose one or combine):

1. **Assignment-stability (no-change) criterion**:
   * Stop when two consecutive rounds yield identical assignments (every user remains assigned to the same fog node), and prices did not change beyond small epsilon.
   * Implementation:
   * prev\_assignments = None
   * prev\_prices = None
   * for round\_idx in range(MAX\_ROUNDS):
   * # compute assignments/prices
   * ...
   * # after assignments:
   * if prev\_assignments is not None:
   * if assignments == prev\_assignments and max\_price\_change < EPS\_PRICE:
   * break # equilibrium reached
   * prev\_assignments = assignments.copy()
   * prev\_prices = prices.copy()
2. **Utility-stability (no incentive to change)**:
   * Stop when no user can improve their own utility by unilaterally switching fog nodes given current prices (i.e., a pure strategy Nash equilibrium).
   * Implementation:
     + For each user u, compute current\_utility.
     + For each other candidate fog f2, compute hypothetical price for u if moved to f2 (i.e., recompute demand on f2+u, updated price), and recompute user utility. If any user has an improving move, the system is not at equilibrium.
     + Stop only if **no** user has an improving unilateral deviation. This requires trying O(n\*m) hypothetical checks (n users × m fogs).
   * This is the rigorous game-theory equilibrium test.
3. **Price-change threshold**:
   * Stop if the maximum absolute price change across all fog nodes between rounds is ≤ ε (tiny).
   * Implementation: compute max(abs(price\_new - price\_old)), compare to EPS.
4. **Combined rule (practical)**:
   * Terminate if *both* assignment-stability and price-change threshold hold, or round\_idx >= MAX\_ROUNDS.

**Recommendation for implementation (practical and correct):** use the combined rule (#1 + #3) in simulation because #2 is computationally expensive but the true Nash check provides theoretical guarantee.

**Time complexity & performance notes**

* **Algorithm 1 (Main loop):** if |list| = n (number of collected demands) and resource components k constant (3), inner check is O(k) per demand; so roughly O(n) for full pass if using transactional reservation. If you do time-slot checks (reserve per time unit), cost grows by number of time slots T: O(n \* T).
* **Algorithm 2 (Price Calculation):**
  + Warm-up (rounds ≤ W): O(1).
  + Otherwise:
    - Retrieving samples s — depends on how many windows q you choose and sample size m.
    - process(s) (computing similarity) is O(m \* k).
    - Summations for Qi are O(m).
  + So price calc per agent worst-case O(m). If m ~ n, worst-case O(n).
* **Equilibrium check (rigorous Nash)**: O(n \* m) hypothetical price recomputations (m fogs). This can be costly for large simulations — choose parameters wisely.

**Concrete testing and integration checklist (what to run now)**

Use these steps to test the full system end-to-end:

1. **Populate demand\_store** with synthetic requests (Days 15 & 19).
2. **Start simulation** at Tr = now and choose x = 30 (lookback window).
3. **Collect list**, shuffle it (l ← rand(list)).
4. For each i in l:
   * Snapshot fog capacities (or use transactional update).
   * Run feasibility check loop (lines 10–16). If fail use reset() semantics.
   * Compute P\_i = Price(...):
     + If rounds <= W use P\_hat\_i.
     + Else gather s, compute L\* and L\*\*, compute Qi.
     + If Qi <= gamma\_i accept; otherwise reject.
   * If accepted, commit capacity reservation and record assignment & price.
5. **After one pass**, update rounds++, recompute prices over fog nodes if necessary (global demand changes).
6. **Iteration**:
   * Repeat above full pass until stopping condition met (Day 25).
   * Log per-round prices, per-user utilities, acceptance rates.
7. **Validation**:
   * Visualize price evolution per fog node (we already did for Days 23–24).
   * Visualize user utilities and provider utilities per round.
   * Compare baseline (fixed price) with dynamic (GTRADPMFC) and FogPrime (if available).

**Parameter recommendations (practical)**

* LOOKBACK\_WINDOW (x): 30s–300s (simulation units) — choose based on scenario density.
* W (warm-up rounds): 2–5 for stability.
* delta (δ): try {0.0, 0.1, 0.2, 0.3} — paper uses these.
* sim\_threshold for splitting L\*: either top-k (k=3..5) or distance threshold tuned by validation.
* EPS\_PRICE (stopping threshold): 1% of average base price.
* MAX\_ROUNDS: 30 or 50 to avoid infinite loops.

**Summary — how everything ties together with what you’ve done**

* Your earlier code (Days 15–24) already produced user requests, fog nodes, fixed pricing, a simple dynamic pricing rule, utility functions, and an iteration loop.
* The uploaded pseudocode details how to operationalize a **robust, history-aware** dynamic pricing mechanism:
  + Collect historical samples from (Tr-x, Tr).
  + Process samples into L\* (similar) and L\*\* (dissimilar).
  + Compute Qi from Equation (3) and accept only if Qi <= γ\_i.
  + Use warm-up threshold \hat P\_i for a few rounds W to stabilize bootstrap behavior.
  + Repeat iterations with **transactional capacity reservation** and stopping criterion (Day 25).
* I gave exact code patterns (transactional reservation, process(s), compute\_Qi) you can paste into your simulator and connect to previous Day scripts (request generation, fog node class, plotting functions).