Trust Aware Scheduling of Online Tasks in FoG Nodes

Tanisha Salvi (210101103) / Sreehari C (210101101) / Rashaad Ali Baig (210101085)

April 19, 2024

Abstract

We propose a heuristic solution for task scheduling in dynamic Fog computing environments with evolving trust and reliability requirements. Our approach integrates trust dynamics, reliability constraints, and cost minimization. Tasks are scheduled on homogeneous Fog nodes considering individual and shared trust metrics, dynamic task failure probabilities, and non-linear cost functions. By dynamically adapting to changing conditions, our heuristic involves usage of rolling horizon which involves continually updating and refining scheduling decisions. Extensive simulations and graph plots for different parameters validate the effectiveness of our approach in real-world scenarios.

1 Introduction

The Trust Aware Scheduling of Online Tasks in Fog (FoG) Nodes is a pivotal area of research aimed at orchestrating task execution while considering both the reliability requirements of tasks and the trustworthiness of processing nodes. This problem is particularly pertinent in scenarios where tasks arrive continuously, each with its unique set of constraints and deadlines.

This paper explores the intricacies of Trust Aware Scheduling in Fog Nodes, proposing novel approaches to optimize task allocation while ensuring adherence to reliability constraints, deadline requirements, and cost considerations. By leveraging trust metrics and real-time feedback mechanisms, the proposed framework aims to enhance the overall efficiency and dependability of Fog-based task execution, thereby facilitating the seamless integration of edge computing into diverse application domains.

2 Problem Statement

We are given a stream of tasks in an online manner, where each task is associated with a user from a total of U users. The objective is to devise an approach that minimizes cost while scheduling maximum number of tasks.

2.1 Task Parameters

- arrival_time (A_i): The time at which the task i arrives.
- execution_time (P_i): The time required to execute task i.
- user (U_i): The user associated with task i.
- relative_deadline ($d_i = a_i + K$): The relative deadline of task i, where K is constant and $K > P_i$ for all tasks.
- reliability_requirement (R_i): The reliability requirement associated with task i.

2.2 Fog Node Parameters

The underlying system consists of M homogeneous Fog/processing nodes.

- shared_trust ($ST[m]$): The shared trust (average user rating or brand value) of Fog node m.
- individual_trust ($DT[u][m]$): The individual/direct trust from user u to Fog node m. Initially, all user ratings are 0.5.
- failure_rate ($F[m]$): The failure rate of Fog node m, which changes over time.

2.3 Cost Function

The cost of unit execution time on a node is determined by the shared trust value of the node. The cost function is non-linear and can be represented as:

$$C = \text{Base} + BR \times ST[m]^2$$

where Base is around 30% of the brand constant BR.

2.4 Task Failure Probability

The failure probability of a task on a node depends on its execution time and the failure rate of the node. It can be modeled as:

$$FP = 1 - \exp(-f \cdot t)$$

where f is the failure rate of the node and t is the execution time of the task.

The problem involves scheduling tasks in a dynamic Fog computing environment to minimize cost while meeting reliability and deadline constraints and maximising number of tasks scheduled.

3 Assumptions

- The simulation involves generation of jobs at discrete units of time (1 unit). The schedule changes dynamically at these discrete units.
- The failure rate f corresponding to each fog node is updated after a fixed numbers of tasks have been successfully executed on that fog node. The value is assigned randomly.
- The time t in the failure probability formula
- Deadline penalty is Considered Uniform

$$FP = 1 - \exp(-f \cdot t)$$

  is taken to be the execution time of the task. Whether a task succeeds or not, is known only after the execution.
- The execution time, user assigned and reliability requirement for each job is generated randomly from uniform distribution. The number of jobs arriving at each time unit is also random.

4 Solution Approach

4.1 Key Observations

- Jobs with earlier deadline should be scheduled first, since the job gets more attempts in case of failure with higher chances of completion via **stays ahead** argument.
- In case of same deadline, shorter jobs should be given priority as more jobs can be scheduled to execute.
- For any task, schedule it on the fog node which "just" meets its reliability requirement to minimize cost. Scheduling a task on fog node with minimum failure rate tends to increase the shared trust of a single machine since most jobs will be scheduled on it (Shown through data). This will further impact the cost function (which is proportional to the shared trust of a fog node). Hence, scheduling on the basis of minimum failure rate does not seem to be a proper fit.
- Scheduling shorter jobs strongly opposes longer jobs as it has less cost and less chances of failure (as both depend on execution time).
- When two jobs can't be scheduled together on a single node and one of them misses a deadline in any way then it is better to remove the job with larger execution time from the schedule. Proof using exchange argument: if shorter job is removed then it leads to more cost (as execution time is more) but even more chance of its own failure (as latency of both jobs are same, it only depends on execution time) and failure of jobs coming after it (as it shortens their latency).

4.2 Detailed Heuristics

We Assume that some Random Number of Jobs Arrive in each interval of 1 second and they have Random Execution Time, Reliability Requirement All of which are Chosen from some Uniform(0,a) Distribution

So Whenever a New Job Arrives We Run Our Heuristic Along With Rolling Horizon As Follows:

1. Firstly, we select all the machines that meet the jobs' reliability requirements and iterate over them in ascending order. Among the possible machines, we consider the machine on which the job will run without violating deadline constraints. To serve this purpose, a **global scheduler** is used.

2. Now, we try to generate a new schedule for that Fog Node for this task while also including the already scheduled but not yet started execution jobs. Each Fog Node is associated with a **local scheduler** which considers a task's **expected time of execution** during scheduling. We observe that this boils down to a problem of $1|p_j, d_j| \min U$, which can be solved optimally through Algorithm 1.

Algorithm 1 : $1|p_j, d_j| \min U$, **Local Schedule Generator**

```
Input: List of tasks with arrival time $A_i$, execution time $P_i$, and relative deadline $d_i$.
Sort tasks in ascending order of $d_i$.
$S := \emptyset$; $t := 0$;
for $i := 1$ to $n$ do
  $S := S \cup \{i\}$;
  $t := t + p_i$;
  if $t > d_i$ then
    Find job $j$ in $S$ with the largest $p_j$-value;
    $S := S \setminus \{j\}$;
    $t := t - p_j$;
  end if
end for
```

3. The local scheduler might eject out a previously assigned task (which has not started its execution yet), when a new task arrives (with lesser expected time of execution) in case only one out of these two tasks meet deadline constraints. Note that the execution time of ejected task is more than the execution time of the new task. Instead of dropping this ejected task, we store it in the **global scheduler** (in view of future possible schedule - **Rolling Horizon**). This process will make sure that under heavy contention of machines, only those tasks tend to be unfavoured for scheduling which have maximum execution time, instead of tasks being scheduled FCFS way (which leads to tasks farther deadline away being dropped). Since a task's failure rate depends on its execution length and favoring shorter jobs instead of longer jobs works in our favor as it reduces the cost (per unit time), maximizing probability of success (as failure probability is directly proportional to length of tasks) and leaving more latency hence more remaining time for tasks in future.

4. Note that this ejected and re-entry in global scheduler does not produce a cycle (infinite loop) at any discrete time of scheduling. If the task cannot be scheduled on any node (being the longest job, ends up getting removed from the schedule - **halting condition**), we exclude it from being scheduled according to **local scheduler** and schedule it in a different way.

5. We repeat this until the **global queue** is empty

6. For every task that was excluded (unability to meet deadline requirements as per expected time):

   We again select all the machines that meet the task's reliability requirements and iterate over them in ascending order and now schedule it on that node where there is maximum probability of success based on geometric success (getting k successes in x trials).

7. If we still can't schedule, then we try to schedule it assuming that all jobs are successful in the first try and try to schedule this job on that machine which gives maximum latency for this job.

8. Both of the above metrics are chosen in view of latter half of the optimization goal - minimizing the number of unscheduled jobs.

9. In case of any success, we remove the task from global queue and schedule it in the corresponding local queue.

10. In case the job is still unscheduled, it is still kept in the global queue so that in future:

   (a) it may so happen that its reliability requirements are met and it gets scheduled (similar to **rolling horizon**)

11. We definitely drop a job if it leads to passing a deadline as we have assumed deadline to be hard.

Algorithm 2 Global Scheduling Algorithm

```
Input: global_queue, global_scheduler, local_schedulers, reliability_requirement
repeat
  while length of global_queue > 0 do
    Iterate over Fog nodes in ascending order of reliability:
      Calculate reliability requirement satisfaction
    if reliability requirement satisfied then
      Perform Local Schedule Generator on current job and existing local queue
        Obtain unschedulable_jobs
        Put unschedulable_jobs in global_queue
      if current job not in unschedulable_jobs then
        Remove current_job from global_queue
        Break
      end if
    end if
    if current job still in global_queue then
      Iterate over Fog nodes in ascending order of reliability:
        Calculate maximum probability of success for current job in each Fog node ??
        Considering other waiting jobs' expected times in local_scheduler
      if max_probability > 0 then
        Schedule current job temporarily in local_queue of the Fog node with highest probability
        Remove current_job from global_queue
        Break
      else
        Iterate over Fog nodes in ascending order of reliability:
          Calculate maximum latency to deadline for current job in each Fog node
          Considering other waiting jobs' actual execution times in local_scheduler
        if node with latency >= 0 then
          Schedule current job temporarily in local_queue of the Fog node with suitable latency
          Remove current_job from global_queue
          Break
        end if
      end if
    end if
  end while
until no jobs left in global_queue
```

Algorithm 3 Job Scheduling Algorithm

```
Wait for the jobs to arrive in the global_queue at each second.
Call Global Scheduling Algorithm
Call Local Completion
Call Local Scheduler
```

12. In **Local Completion** Once a task's execution time is completed, compute its success probability.

13. If the task is successful, update trust metrics, mark the task as completed, and clear the active fog node flag.

14. If the task fails, update trust metrics and reschedule the task on the same machine or mark it as dropped if it exceeds the deadline. The node is assumed to be busy until the maximum of execution time or expected time, whichever is longer. Updating the cost based on the execution time, fog node's shared trust, and other constants is done accordingly.

Algorithm 4 Local Completion Algorithm

```
1: Input: List of fog nodes with active tasks.
2: for each fog node $node$ in $constants.fog_nodes$ do
3:   if $node$ has an active task AND it's execution time is complete then
4:     Calculate the success probability based on the node's failure rate and task's execution time.
5:     Update cost
6:     if Success then
7:       Update trust metrics for Success
8:       Clear the active task flag on the node.
9:       if Total count of successful tasks reaches a window size then
10:        Call Failure change to change the nodes Failure Rate Randomly and Empty all the jobs from the node's
           local queue in the global queue to be Scheduled again
11:      end if
12:    else
13:      Update trust metrics for Failure.
14:      if Scheduling the Job again leads to missing it's deadline then
15:        Drop the Task and clear the active task flag
16:        Set the node as idle
17:        Exit the loop.
18:      end if
19:      Update task's expected time and release time for rescheduling.
20:      Set the node as busy until the maximum of execution time or expected time.
21:    end if
22:  end if
23: end for
```

15. After **Local Completion** is complete, in **Local Scheduler** all the Fog Node which are idle, execute the first job in their respective local scheduler. It sets the time the node will be busy to be the expected time of the task so that the **Local Schedule Generator** will work with that time.

Algorithm 5 Local Scheduler Algorithm

```
1: Input: List of fog nodes with local queues of tasks.
2: for each fog node $node$ in $constants.fog\_nodes$ do
3:   if $node$ is idle and $node$ has tasks in its local queue then
4:     Select the first task $task$ in $node$'s local queue.
5:     if Scheduling the current Job leads to missing its deadline then
6:       Mark $task$ as dropped and remove it from the local queue.
7:       Continue to the next iteration.
8:     end if
9:     Set $task$'s release time as the current time.
10:    Set $node$ as busy until the expected completion time of $task$.
11:    Remove $task$ from $node$'s local queue.
12:    Set $task$ as active on $node$.
13:  end if
14: end for
```

4.3 Time Complexity Analysis

At each second, we are sorting the global queue or using a priority queue at the global scheduler, which will cost us $\mathbf{O}(n \log n)$ (assuming n is the number of jobs). Now for each local queue, we are sorting it once for a job, which will cost us $\mathbf{O}(mn \log n)$ per job, i.e., $\mathbf{O}(mn^2 \log n)$ total. All other operations are linear in n. Thus, the overall time complexity remains at $\mathbf{O}(mn^2 \log n)$.

5 Parameter Analysis

Let's consider the case with: **K** = 20, Number of Jobs coming per second **N** U (0,6), Execution time of each job **T** U(0,10) ,Number of processors **M** = 10, failure rate of each node **f** U(0,0.2). We will run our algorithm for 60 seconds, i.e deadline of last job is 80. So while plotting these when we change some parameters the others are assumed to be same as given Case.

5.1 Successful tasks Analysis

- The Final Cost is calculated as the weighted sum of cost and deadline misses. The normalized cost is calculated by dividing the actual cost by the maximum cost, which is represented as $(Base + BR) \times$ Max_execution_time. The Final Cost formula can be expressed as:

$$\text{Final Cost} = \frac{\sum_{i=1}^{n} \text{Normalized Cost}_i + \text{Dropped Tasks}}{\text{Total Number of Tasks}}$$

  Where:

  - Normalized $\text{Cost}_i = \frac{\text{Cost}_i}{(Base+BR)\times\text{Max execution time}}$
  - Dropped Tasks represents the number of tasks dropped or missed.

- In Figure 1, you can see that increasing the number of nodes beyond a certain point doesn't increase the success rate of jobs beyond a certain value. We can show this through simple expected time analysis and probability.

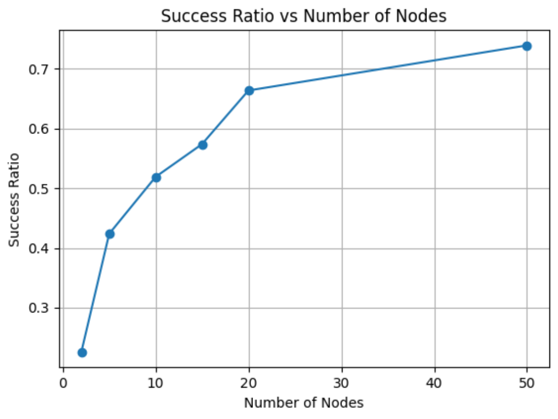
  In the maximum case where the number of nodes approaches infinity, each job will get its own machine for itself, and the latency for each job will be equal to the relative deadline K. So, the probability of the job of length T being successful in K time can be calculated as follows.

  Defining a success event when a task successfully executes, this event will have probability e^{-fT}, where f is its failure rate and T is the execution time of the job.

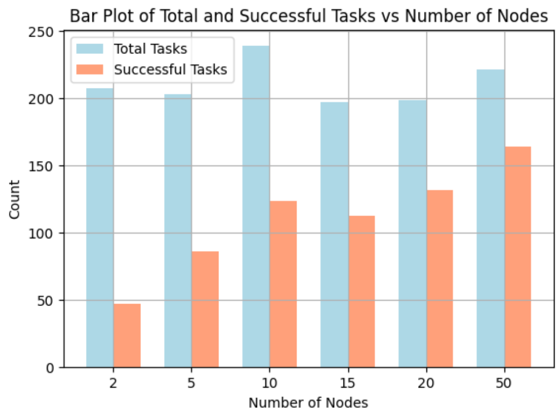
  So, the number of times this event can be played is $t = \lfloor K/T \rfloor$. Thus, the probability that this success event doesn't occur during t independent samples is (Probability of failure)^t, and the probability that success occurs during these t tries is:

$$1 - (\text{Probability of failure})^t = 1 - \left(1 - e^{-fT}\right)^{\lfloor K/T \rfloor}$$

  which we need to average over t from 0 to 10 and f from 0 to 0.2. The value comes out as 0.84, which is close to what we are achieving with 50 (0.74) nodes, and this value keeps on increasing with increasing number of nodes in the system.



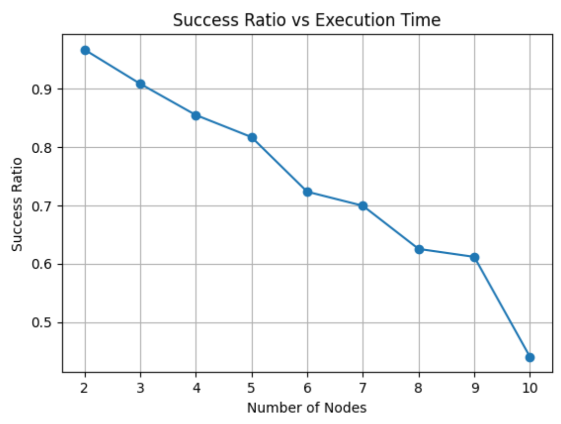
((a)) Success Ratio vs No. of Fog Nodes



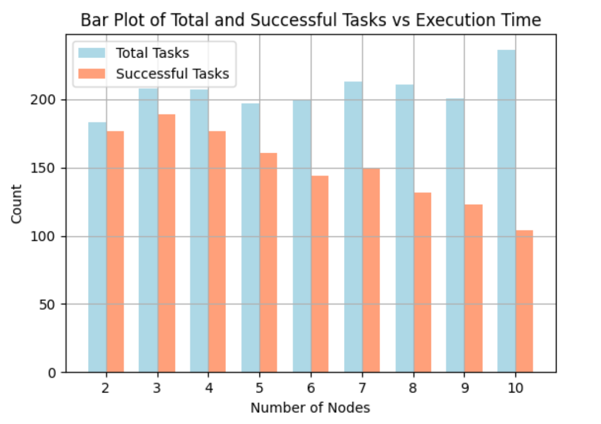
((b)) Total/Successful tasks vs No. of Fog Nodes

Figure 1: Plot comparison varying Number of Fog Nodes

- The next 3 of the plots are pretty intuitive as execution time or number of task increases the success ratio decreases. And as deadline is increased success ratio increases as jobs get more latency to get scheduled

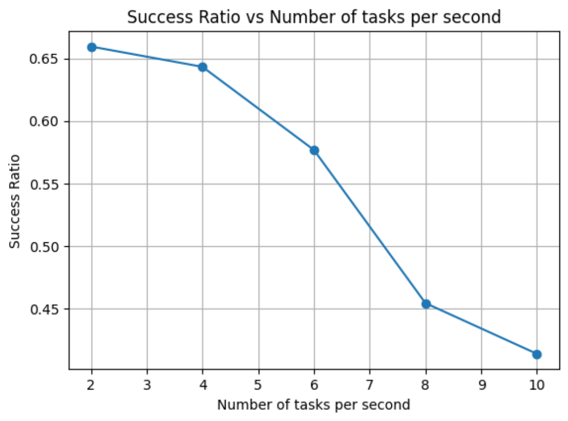


((a)) Success Ratio vs Execution Time

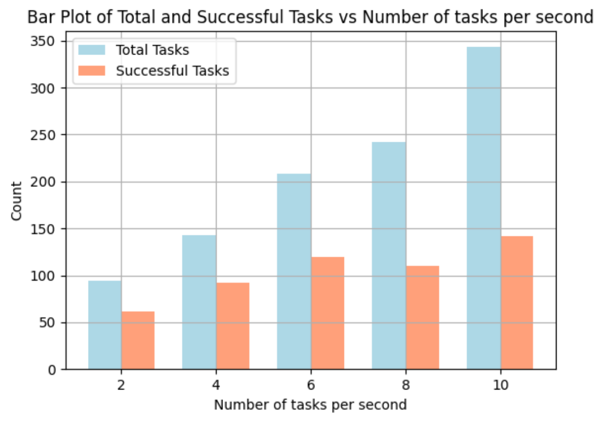


((b)) Total/Successful tasks vs Execution Time

Figure 2: Plot comparison varying Execution Time



((a)) Success Ratio vs No. of tasks per second



((b)) Total/Successful tasks vs No. of tasks per second

Figure 3: Plot comparison varying No. of tasks per second

5.2 Cost Analysis

Let's consider the case with: **K**=20, Number of Jobs coming per second **N** U (0,6), Execution time of each job **T** U(1,10) ,Number of processors **M** = 10, failure rate of each node **f** U(0,0.2). Let's run our algorithm for 60 seconds, ie deadline of last job is 80. We will here try to prove on the expected bound of the number of jobs that can be scheduled by any algorithm mathematically, Here as the second component of the cost that occurs due to **ST** is anyway minimized in our algorithm by **stays ahead** argument, we need to focus only on the first part that occurs due to unscheduled jobs.

5.2.1 Considering No Failures:

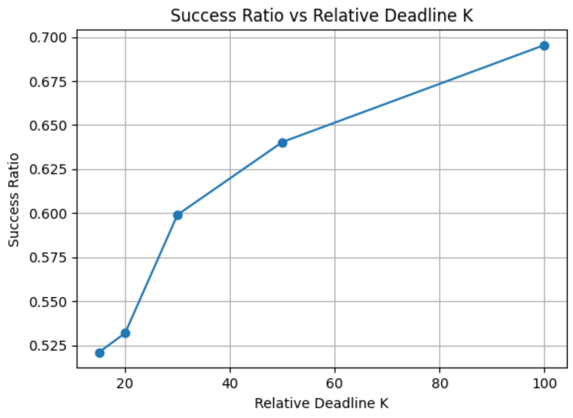
Consider a case where no failures occur, that is, we assume all the machines have **zero failure rate**. Then,

$$E[N] = 3$$
$$E[T] = 5$$
$$\text{Total Expected time for scheduling} = E[N] \times E[T] \times 60 = 5 \times 3 \times 60 = 900$$
$$\text{Total time available} = 60 \times 10 = 600$$

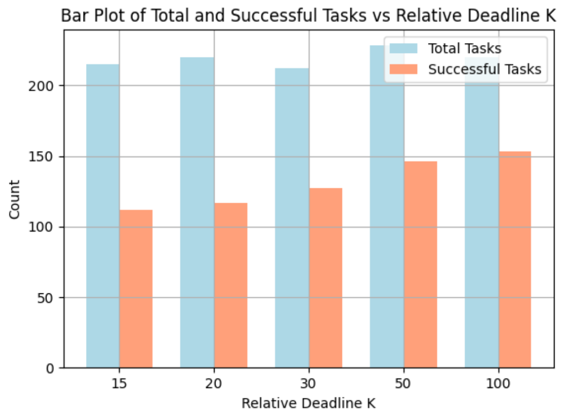
To find the fraction of time scheduled:

$$\text{Fraction of time scheduled} = \frac{\text{Total time available}}{\text{Total Expected time for scheduling}} = \frac{600}{900} = 0.6666\ldots = 66.67\%$$

Therefore, even in the best-case scenario where none of the jobs fail, only approximately 66.67% of the scheduled time can be utilized.That is 66.67%of the jobs can be scheduled as each job has atleast unit execution time.

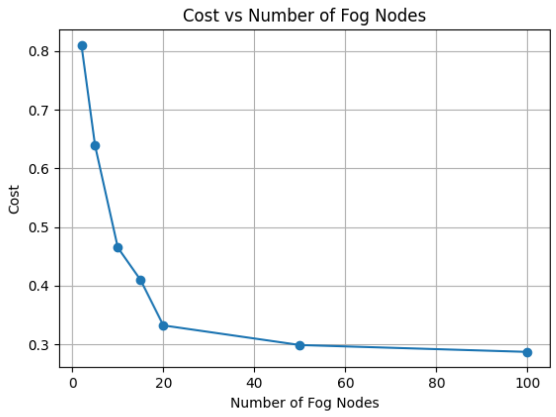


((a)) Success Ratio vs Relative Deadline K

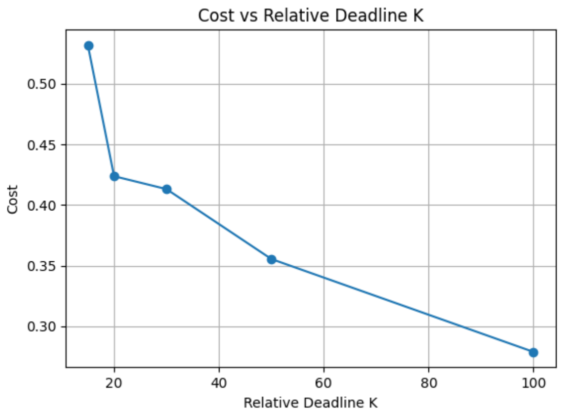


((b)) Total/Successful tasks vs Relative Deadline K

Figure 4: Plot comparison varying Relative Deadline K



((a)) Cost vs Number of Fog Nodes



((b)) Cost vs Relative Deadline K

Figure 5

5.3 Considering minimal failures

Here we consider minimum expected extra time required in case of failure and analyse on it to produce the best case expected upper bound.

Let **X** be the execution time of the current job. If it is scheduled in a node with failure rate **f**, the expected execution time can be calculated as follows:

The success probability is given by:

$$\text{Success probability} = e^{-fX}$$

Since we schedule the node until success, it follows a geometric distribution with parameter $p = e^{-fX}$. The expected value of X under this distribution is:

$$E[T] = Xe^{fX}$$

To find when $E[T]$ is minimized, set $X = 1$. The failure rate is minimized when $X = 1$, meaning the extra fraction of time taken due to failure is minimized:

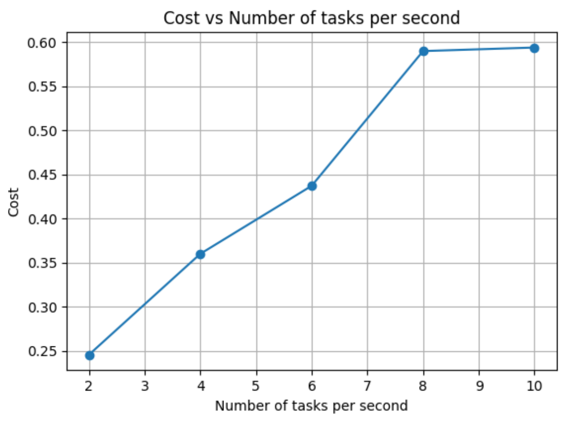
$$E[\text{Extra fraction of time taken due to failure}] = E[e^f] = \int_0^{0.2} \frac{e^f}{0.2}\, df = 1.105$$

Therefore,

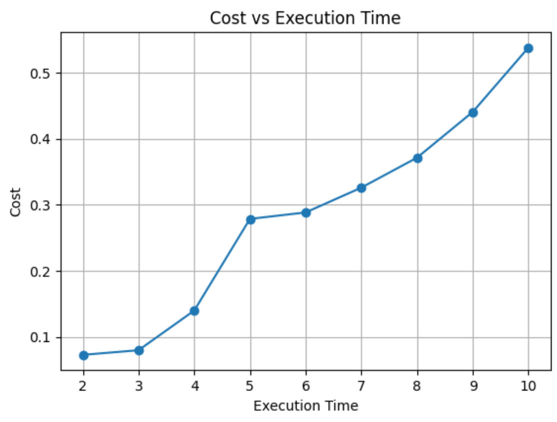
$$E[T] = X \times 1.105$$

Now, take the expectation of this with respect to X:

$$\text{Expected time} = 5 \times 0.221 = 1.105$$

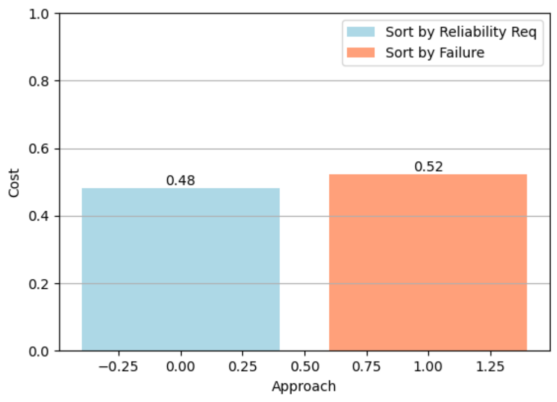


((a)) Cost vs Number of Tasks per second

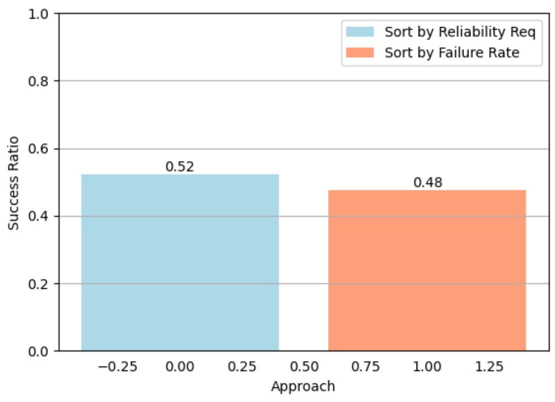


((b)) Cost vs Execution Time

Figure 6



((a)) Cost comparison



((b)) Success Ratio comparison

Figure 7: Validating sorting fog nodes through trust values.

Hence, the expected time is approximately 1.105. This is the minimum expected time taken by a job considering minimum failure. Thus as before,

$$E[N] = 3$$

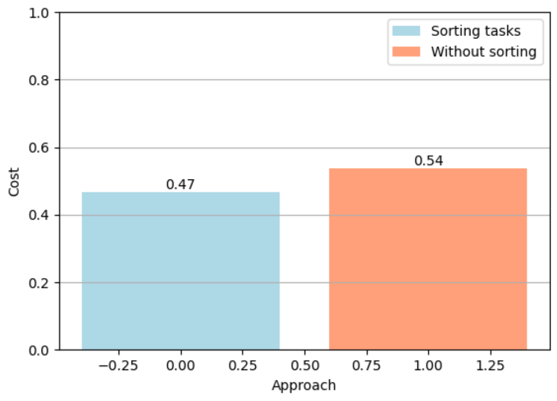
$$E[T] = 5 \times 1.05 = 5.5 \text{ Total Expected time for scheduling } = E[N] \times E[T] \times 60 = 5.5 \times 3 \times 60 = 990$$

Total time available $= 60 \times 10 = 600$

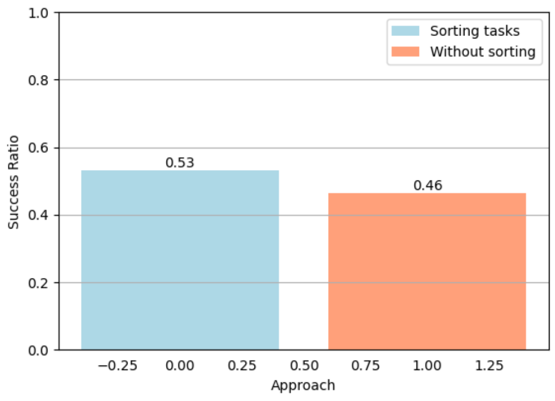
To find the fraction of time scheduled:

$$\text{Fraction of time scheduled} = \frac{\text{Total time available}}{\text{Total Expected time for scheduling}} = \frac{600}{990} = 0.60\ldots = 60.06\%$$

Therefore, even in the best-case scenario where we consider the minimum failure rate of the jobs , only approximately 60.67% of the scheduled time can be utilized.That is 60.67%of the jobs can be scheduled as each job has atleast unit execution time.



((a)) Cost comparison



((b)) Success Ratio comparison

Figure 8: Validating sorting jobs through deadline and execution time values.