mathematics

MDPI

Article

Prediction-Driven Non-Preemptive Scheduling with Bounded Overtaking: A Per-Job Wait Guarantee

Yazhou Guo [1,*], Zexin Lin [2], Chengyang Huo [3] and Yurong Song [1]

[1] College of Automation, Nanjing University of Posts and Telecommunications, Nanjing 210023, China; songyr@njupt.edu.cn
[2] College of Artificial Intelligence, Nanjing University of Posts and Telecommunications, Nanjing 210023, China; b25060222@njupt.edu.cn
[3] Division of Logistics and Transportation, Shenzhen International Graduate School, Tsinghua University, Shenzhen 518055, China; huocy24@mails.tsinghua.edu.cn
* Correspondence: gyz@njupt.edu.cn

Abstract

Before a programming deadline, many students' programs queue for a few grading servers. Ordering them by predicted running time cuts the mean wait, but known bounds with predictions hold on average or against an optimum, not per job. We prove an identity for k non-preemptive, work-conserving servers with time limit L. A job's extra wait over first-come first-served (FCFS) equals the net running time of jobs that overtook it, divided by k, up to a tight error of (2 − 2/k)L. A guard around any scheduler caps the work that passes each job, so every job stays within a promise G, whatever the arrivals and predictions. Its additive constant (3 − 2/k)L is exact, and completion reports delayed by δ add at most L + δ. On one test term, gradient-boosted trees, a recurrent network and a heterogeneous graph neural network predicted running times about equally well. On three course terms withheld until the method was frozen, a guard promising 600 s kept 0.894 (class-term interval 0.116–0.933) of shortest-job-first's reduction in the simulated 99th-percentile wait before deadlines at the busiest load. The gain shrank above full load. On a second platform, run at twice its target load, the promise held but the gain did not appear.

Keywords: bounded overtaking; distributed systems; first-come first-served; heterogeneous graph neural network; multiserver queues; non-preemptive scheduling; per-job guarantee; resource allocation; runtime prediction; scheduling with predictions

MSC: 90B35; 68M20; 60K25; 68W27

Academic Editor: Frank Werner

Received: 28 September 2026
Revised: 29 October 2026
Accepted: 1 November 2026
Published: 3 November 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

1. Introduction

Ten minutes before a programming assignment is due, the grading servers of a university course receive submissions in a burst, and a queue forms. Each program is compiled, run against the test cases and stopped when it reaches a fixed time limit [1], and each student watches a position in a queue. A program that runs for one second can wait minutes for its result, and the students who submit closest to the deadline wait longest. A similar queue forms in other shared execution services. A developer who pushes code before a release date waits for the continuous-integration (CI) machines to run the test suite, and a user of a cloud platform waits while a burst of function calls is served. In each

case many users share a fixed set of k machines, a job runs without interruption once it starts, and no job can run longer than a common time limit L. Such a pool is a small distributed system. Its servers are separate machines, they report completions to the scheduler over a network, and the scheduler itself can be replicated. This paper asks what such a service can promise each of its users at its busiest hour.

Two features of the demand make the service slowest when it is busiest. Submissions follow external calendars, such as coursework deadlines and release dates, so they arrive in bursts. People concentrate their work before a deadline [2], and conference registrations crowd into the days before theirs [3,4]. A small fraction of the jobs also carries a large share of the work. In the log of cloud-function calls that we use [5,6], the longest 1% of the calls account for 44% of the total running time. In the two logs of students' programming submissions, the longest 1% account for 37.5% and 27.5%. The university grading system shows both features clearly, and it is our first case. Peveler et al. replayed the busiest four hours before a deadline four and sixteen times faster on one grading server. The container-based grader then used all of its concurrent slots, and its waits fluctuated, with up to almost 25 s more waiting than a jailed sandbox [7]. The manual of a programming-contest judging system recommends one judging machine per 20 teams and notes that with fewer machines teams may wait longer for their results [8].

Ordering the queue by predicted cost exploits the second feature, as a queue of three jobs on one server shows (Figure 1). Job A loops until the 60 s limit stops it, and jobs B and C take 0.2 s each. In arrival order, the mean wait is 40.1 s. With the shortest job first, it is 0.2 s, at a cost of 0.4 s to A. This arithmetic assumes that the scheduler knows before A starts that A is long. The ordering also has a second cost, which arises even when every estimate is right. A long job waits while shorter jobs that arrive after it keep starting ahead of it. In our simulation of the busiest period that we tested, shortest-job-first (SJF) with exact running times made one job wait 4755.5 s longer than it would have waited under first-come first-served (FCFS) order. That is 79.3 times the time limit (Section 8.4).

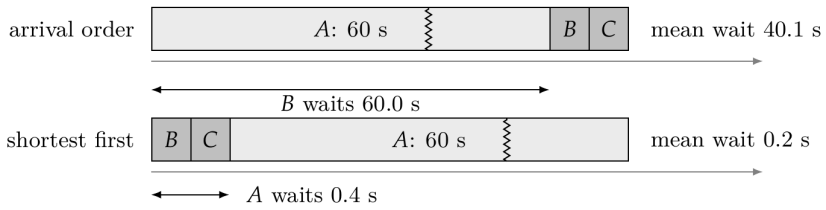


Figure 1. The three-job example of Section 1 on one server: A is stopped at the per-job limit $L = 60$ s, B and C take 0.2 s each, and all three are present at $t = 0$. The boxes are schematic, not to scale, and the zigzag marks the elided part of A. The two orders execute the same total work. Reordering takes 60 s of waiting off each of B and C and puts 0.4 s onto A.

A line of work on algorithms with predictions bounds how an algorithm performs as a function of the error of its predictions [9,10]. For scheduling, these bounds compare the total or mean completion time with that of an optimal schedule [11–14]. An operator, however, is asked about individual jobs, and a predictor in service can be wrong by any amount. Two results come closer. One bounds the worst slowdown of any job on a machine that can interrupt jobs [15], and one optimises the asymptotic tail of the response time when arrivals are Poisson and job sizes are light-tailed [16,17]. We have found no guarantee that is stated for every job against FCFS, holds for any arrival sequence and any predictions, allows no interruption, covers k servers, and can be enforced around any scheduling rule. This paper answers three questions.

1. Can a job's running time be estimated when it arrives, from the history available at that moment?

2. How should a scheduler that cannot interrupt jobs order the queue once it has such estimates?
3. What can the service promise every job when the estimates are wrong, and when the completion reports that it relies on arrive late?

For the third question we change the reference point instead of assuming more about the predictions. For each job we bound how much longer it waits than it would have under FCFS on the same sequence of submissions, with no assumption about the arrivals or the predictions. The mechanism is an overtake budget. For every waiting job, the scheduler adds up the running time of the jobs that arrived later but were started ahead of it. Once that total reaches the job's budget, the job is served before any later arrival. This guard sits on top of any scheduling rule, including one driven by a learned predictor, and changes only which job starts next. A cap $B_{\max}$ on the budget fixes the promise. The budget counts running time rather than the number of jobs that pass, because of an identity that we prove. Consider any rule that never leaves a server idle while jobs wait and never interrupts a job. Under such a rule, a job's extra wait relative to FCFS equals the net running time of the jobs that overtook it, divided by k, up to an error of at most $(2 - 2/k)L$. Completions reach the guard as reports from the servers, so we also bound its promise when these reports cross a network and arrive late.

For the first two questions, we compare three kinds of predictor on the main grading log. We train and test all three on the same data, under the same rule about what can be seen when a job arrives. The first is gradient-boosted decision trees [18]. The second is a gated recurrent unit (GRU) [19] that reads each user's earlier jobs. The third is a heterogeneous graph neural network (HGNN) of the relational graph convolutional network (R-GCN) type [20], over users, programming tasks and assignments. The history available at arrival separates long jobs from short ones on both grading logs (Section 4). The graph's edges add no accuracy that our intervals can detect. Adding them to the same network without edges changes the area under the receiver operating characteristic curve (AUROC) by less than 0.002. On tasks with fewer than five earlier results, the neural models rank jobs better than the trees. However, the network without edges does nearly as well, so this gain comes from the neural model of each job's own features. The scheduler orders jobs by their expected running time given the features. For this score, treating a long job as short is the costlier mistake in every pool built from the main log and at every load that we tested (Section 8.3).

This study makes three contributions.

1. We prove the identity above and show that its error constant $(2 - 2/k)L$ cannot be improved. In the other direction, any per-job promise of this kind limits the running time of the jobs that overtake a job, up to an additive $O(kL)$. A limit on how many jobs overtake it is not required. Such a count gives a promise only if every pass is charged the worst case L, which on our main log is 150 times the mean running time (Section 6.4).
2. We give the guard and prove that it keeps every job's extra wait over FCFS below $B_{\max}/k + (3 - 2/k)L$, for k servers, any arrivals and any predictions, and that neither constant can be improved. An operator who wants to promise G seconds sets $B_{\max} = k(G - (3 - 2/k)L)$, which needs no data. The same promise keeps the largest flow time below $(3 - 2/k)$ times its optimum plus G. Completion reports that arrive up to δ late enlarge the promise by $L + \delta$, and by nothing if each server reports its own completion before it takes its next job. If replicas of the scheduler share the dispatch history and hold the same reports, they agree on which jobs the guard must serve next. We also bound the absolute wait from the arrival record alone, and compare the guard with a maximum waiting time, which gives a promise of the same form (Section 6).

https://doi.org/10.3390/math14214020

3. We evaluate the method on a shared pool built from the main grading log, which records how long each run took but not how long it waited (Section 7.2). We check where the method applies on six logs in all. Two come from university systems that grade students' programming submissions, CodeBench and ACcoding [21,22], one from a cloud-function platform [5], two from CI machine pools [23,24] and one from an industrial compute farm [25,26]. The proved bound is checked on every simulated job, and no job exceeds it. In the rest of this paragraph, the scores follow the simulated queue (Section 8.1). A job's score uses an earlier result from its own class and term only once that queue has completed it. Tables and figures that read the recorded times instead say so. SJF, which knows every running time, reduces the 99th-percentile wait before deadlines relative to FCFS. We held out three academic terms of the main log until the method and its parameters were frozen. On these terms, a guard that promises 600 s keeps 0.894 of that reduction at the busiest load (Section 8.2). On the development terms, the same guard keeps 0.705. Resampling whole classes and terms widens the interval of this figure to include zero (Section 7.2). The same resampling on the sealed terms, run after their result was known, gives an interval of [0.116, 0.933] around 0.894 (Section 8.2). On the main log, the gain shrank once the busiest hours exceeded full load, first at the tighter promises (Section 8.8). The gain also needs a long FCFS tail and concentrated costs, and Section 8.7 shows logs that lack them. On the held-out part of the second grading platform, the guarantee held for every job, but the gain did not carry over, in a run whose load reached about twice its target (Section 8.2).

Section 2 reviews related work, and Section 3 states the model and the measures of performance. Section 4 compares the predictors, Section 5 describes the scheduler and the guard, and Section 6 proves the identity and the per-job bounds. Section 7 describes the logs and the shared pool, Section 8 reports the experiments and Section 9 states the limits of the theory. Three parts concern the distribution of the pool. Section 6.8 bounds the guard when completion reports arrive late and when the scheduler is replicated, and Sections 8.9 and 8.10 measure late reports and the cost of the accounts. The physical run of Section 8.8 used two worker processes on one machine. Longer proofs are in the Supplementary Materials.

2. Related Work

2.1. Scheduling with Predictions

In the framework of Lykouris and Vassilvitskii [9], algorithms with predictions are judged by two properties [10]. Consistency is how well they do when the predictions are exact, and robustness is how well they do when the predictions are arbitrarily wrong. In scheduling, predicted sizes have been used to minimise the total completion time of jobs whose sizes are unknown [11]. Learned machine weights have been used to assign jobs online so as to bound the makespan [27], and a predicted order of the jobs has been used in place of their sizes [12]. Queueing models treat the prediction as a random variable [28] or bound its error multiplicatively [13], and online analyses measure the error as a distortion [29,30]. Further work covers many-server limits [31], the cost of obtaining predictions [32] and the worst slowdown of any job on a machine that can interrupt jobs [15]. Mitzenmacher and Shahout survey this line [14]. These results bound a mean or a competitive ratio against an optimal schedule, and the bound degrades with a measure of the prediction error. In their terms, our guarantee is a robustness statement made for each job and on every sample path. It holds for any predictions, but it bounds the loss against FCFS and does not by itself promise a gain. Section 8 measures the gain.

Recent work extends these models in several directions. Predictions can cover only some of the jobs [33]. With size estimates of unknown distortion μ, an online algorithm on one machine is $O(\mu)$-competitive for the total flow time, which is optimal up to a constant by an earlier lower bound [30,34]. When all jobs arrive at time zero and the error is multiplicative, of size η, predicted shortest-job-first on one machine is η^2-competitive [35]. For stochastic jobs, the Gittins policy can cost $\Omega(n)$ times the optimum when the predicted size distributions are only close to the true ones. In contrast, a robust variant stays within a factor polynomial in that closeness [36]. The same policy can also be built from samples of the size distribution with a bounded loss in the mean response time [37]. When the mean sizes of job types must be learned, some policies without preemption incur an excess cost that is sublinear in the optimal cost. However, every such policy incurs an excess of order $n\sqrt{n}$ when two means are close [38]. Suppose that all jobs arrive at time zero and that the predictions can only underestimate the sizes. Then a constant competitive ratio is achievable with a number of preemptions per job that is logarithmic in the ratio of the true to the predicted size [39]. The policy called size estimate hedging freezes the priority of a job once the job's age shows that its estimate was too low, and its authors evaluate it numerically [40]. For servers of large language models, predictions of the remaining output length drive a shortest-predicted-remaining order, which can interrupt a request only early in its run [41]. A virtual token counter bounds the difference in service between any two backlogged clients, and length predictions do not change this worst case [42]. On one such server with a memory limit and no preemption, every deterministic online algorithm has a competitive ratio of order at least $\sqrt{n}$ for the total latency, even with exact output lengths [43]. That work obtains a constant ratio only when all requests arrive together and have prompts of equal length. Suppose instead that the requests arrive together and that each output length is known only to lie in an interval $[\ell, u]$. Then, for a large memory, some policy achieves a ratio of order $\log(u/\ell)$ against a shortest-first schedule with full information, and no policy without the lengths does better [44]. All of these bound a mean, a competitive ratio against an optimum or a difference between clients. However, none of them bounds the wait of each job against a reference order on the same input.

A second line of work wraps a learned policy in a fallback that carries a guarantee. In smoothed online optimisation, an algorithm can follow untrusted predictions and switch to a robust algorithm, so that its competitive ratio stays bounded for any predictions. Such an algorithm can also filter the decisions of a black-box model [45]. In anytime-competitive reinforcement learning, each learned action is projected so that the cost accumulated by every round is at most $(1 + \lambda)$ times that of a prior policy, plus a fixed allowance per round [46]. In online scheduling with covering constraints, a learned scheduler keeps a fairness cost of its decisions over the whole horizon within $(1 + \lambda)$ times that of a fair baseline, for every input sequence [47]. In switch buffers, a safeguard keeps at least $1/N$ of the optimal throughput whatever the predicted drops [48]. In operating systems, guardrails monitor learned kernel policies and fall back to the existing policy after a violation, without a formal bound [49]. Suppose that a dispatcher sends each job on arrival to one of several FCFS servers and knows the exact sizes. Then some dispatching policy is optimal for the mean response time in heavy traffic [50]. This is the setting of Load Balancing Guardrails [51] (Section 2.2), but with FCFS servers. Nearest to our pattern is a real-time job that uses a predicted execution time [52]. The job always meets its hard deadline, and its energy stays at most a fixed factor above that of an algorithm that ignores the prediction, on every execution time. As with our guard, the reference there does not use the predictions and is not the clairvoyant optimum. However, the bound covers the energy of one job rather than the wait of each job in a queue.

2.2. Per-Job Guarantees and Bounded Overtaking

Schwiegelshohn and Yahyapour call a schedule λ-fair when later submissions cannot increase a job's flow time by more than a factor λ. They bound fairness and efficiency together for a policy that interrupts whole groups of processors [53]. Batch schedulers measure a job's delay against a fair start time, obtained by simulating the schedule again without the later arrivals [54,55]. Conservative backfilling gives every queued job a reservation that later jobs cannot delay [56,57]. Slack-based backfilling lets later jobs delay a job only within its slack, which is set when the job arrives and recomputed once [58]. EASY backfilling is the rule of the Extensible Argonne Scheduling sYstem, developed for the IBM SP at Argonne [59]. In a preemptive variant, the scheduler stops and later restarts the backfilled jobs that would delay a higher-priority job [60]. These guarantees rest on reservations or on interrupting jobs. Ours, in contrast, is additive, needs no interruption and is measured against FCFS on the same input.

Bounded overtaking has been studied in two units. The first is a count. Nudge lets a small arriving job swap past the job at the tail of the queue [61]. Nudge-K lets a job of one class pass up to K jobs of another class, while no job is passed more than once [62], and later work refines these policies [63]. In finite-skip models, only the n oldest jobs in the system can receive service [64]. The second unit is work: γ-Boost measures the work of later arrivals that are served ahead of a tagged job [16,17]. These analyses assume stationary Poisson arrivals and target means or tail asymptotics, not a cap enforced on every sample path.

Load Balancing Guardrails [51] are the closest wrapper in architecture. Like our guard, they leave an arbitrary policy free except where a constraint must be enforced. They differ from the guard in the system, in the scheduling at the servers and in the guarantee. First, a dispatcher sends each job on arrival to one of k servers, and each server keeps its own queue. The guard instead orders one central queue in front of k servers. There is thus no dispatch decision for the guardrail counters to restrict. Second, their servers run preemptive shortest remaining processing time (SRPT), and the dispatcher knows the exact size of each job. Our servers never interrupt a job, and the scores can be arbitrary predictions. Third, their theorem assumes Poisson arrivals. It compares the mean response time with that of one SRPT server k times as fast. The ratio of the two tends to 1 as the load tends to 1. The theorem gives no bound for a single job and no comparison with FCFS. Our guarantee bounds the wait of every job against FCFS on the same input, on every sample path. Guardrails thus fit a dispatcher that feeds preemptive servers and knows the job sizes, when the goal is the mean response time under heavy load. The guard fits one queue that feeds non-preemptive servers with predicted sizes, when each job needs a bound on its own wait. Section 8.4 therefore compares the guard with a maximum waiting time and with backfilling. Both order one queue for servers that do not interrupt jobs.

Additive bounds against a reference discipline are standard in packet scheduling [65–68]. The coefficient of Graham's list-scheduling bound [69] also has the same shape as the $(2 - 2/k)$ of Section 6. Table 1 sets the closest methods side by side. Among the methods in that table, only our guarantee bounds the wait of every job against FCFS on the same input on several servers. A plain maximum waiting time also gives such a bound, and Section 6.6 proves it and compares the two. It needs no preemption and holds around any base policy and for any scores. Under Nudge, each job is passed at most once, so on one server Nudge implies a bound of the same form: one small-job length above FCFS. This is our own reading and not a claim of its authors (Supplementary Section S9.2). Supplementary Table S26 adds the information that each method needs, and Supplementary Section S9 compares these guarantees in detail.

Table 1. Settings, performance measures and guarantees of the guard and of the closest existing methods. "Applies to" is the setting in which the guarantee holds, and "Compared with" is the schedule or policy that the guarantee is stated against. $W_P[i]$ is the wait of job i under policy P, k the number of servers, L the time limit (Section 3) and $B_{\max}$ the cap of the guard on the budget (Section 5). M/G/1 is one server with Poisson arrivals and general service times. $L_{\max}$ is the largest packet length and r the link rate. The stretch of a job is its flow time divided by its size.

Method	Compared with	Applies to	Measure	Guarantee
Guard (this work, Theorem 3)	FCFS on the same input	one central queue for k identical servers; any finite arrival sequence; no preemption; any scores	wait of each job	$W_P[i] < W_{\text{FCFS}}[i] + B_{\max}/k + (3 - 2/k)L$ for every job, on every sample path
Predictions, online [10]	offline optimum	one machine, preemption; predicted sizes	total completion time	competitive ratio near 1 for exact predictions and bounded for any predictions
Predictions, queueing [13]	SRPT with true sizes	M/G/1, preemption; estimates within $[\beta s, \alpha s]$ of size s	mean response time	ratio to SRPT bounded by a function of α and β, tending to 1 as $\alpha, \beta \to 1$
Predictions, fairness [15]	offline optimum	one machine, preemption; multiplicative size error	largest stretch	competitive ratio polynomial in the error η and in the ratio P of the largest to the smallest job size, e.g., $O(\eta^3\sqrt{P})$
Safe use of predictions [52]	algorithm that ignores the prediction	one real-time job, hard deadline; speed scaling; predicted execution time	energy	deadline always met; energy at most a fixed factor of the reference's, for every execution time
Preemptive FCFS (PFCFS) [53]	same policy without the later jobs	online releases; rigid parallel jobs, gang preemption	flow time of each job	at most $(2 + 2\bar{s})$ times the reference, where $\bar{s}$ is the preemption penalty relative to the shortest run time, when each job's weight is its number of processors times its run time
Fair start time [54]	start without the later jobs	recorded batch workloads; rigid parallel jobs	start delay of each job	none; a metric computed after the run
Packet-by-packet generalised processor sharing (PGPS) [66]	generalised processor sharing (GPS)	any arrivals at one link; packets not preempted	finishing time of each packet	GPS finishing time plus $L_{\max}/r$
Nudge [61]	FCFS	M/G/1, Poisson arrivals; no preemption	response-time distribution	stochastic improvement over FCFS; at most one swap per job, which on one server implies a wait at most one small-job length above FCFS on every sample path (our reading)
Conservative and slack backfilling [56,58]	reservation made at submission	online submissions; rigid parallel jobs; requested run times	start time of each job	later jobs never delay a reservation, or delay it only within the job's slack
Load Balancing Guardrails [51]	one SRPT server k times as fast (pooled capacity)	a dispatcher that sends each job on arrival to one of k servers with separate queues; Poisson arrivals, heavy traffic; preemptive SRPT at each server; exact sizes	mean response time	for any guarded dispatcher, the ratio to the reference tends to 1 as the load tends to 1; no bound for a single job

2.3. Prediction-Driven Scheduling in Deployed Systems

Backfilling schedulers for high-performance computing replace users' runtime estimates by predictions learned from past jobs [70,71] or learn from past jobs which queue order to use [72]. They report mean slowdown or mean wait. Closest to our first case, Zhang, Wu and Lu predict the running time of submitted programs in a judging queue [73]. They order the jobs by prediction inside fixed-size windows, which limits overtaking by position, and they report the mean response time. Some servers for large language models learn a relative order of the requests [74]. Others target the latency tail, because prediction-driven policies are fragile when the workload shifts and interrupting a request is costly [75]. A third group promotes a request once its wait passes a threshold [76], or once its program has waited longer than a multiple of the program's service so far [77]. Justitia serves the agents that share a server in the order in which they would complete under fair sharing of the server's cache [78]. It bounds how far each agent's completion can fall behind that reference, by a constant built from the largest request and the largest agent. The bound has our additive form, but it holds for Justitia's own order, with no base policy and no budget. Mitzenmacher and Shahout list serving short requests first without starving long ones as an open problem for these servers [14]. Serverless platforms approximate shortest remaining time first from user space [79] or learn a priority for each function [80]. Apart from Justitia, none of these states a bound for individual jobs. A maximum waiting time does carry one, of the same form as ours (Section 6.6), and our guard can be placed around any of these schedulers.

2.4. Deadline-Driven Load

People concentrate their work before a cutoff [2], and conference registrations grow faster as their deadline approaches [3,4]. When the arrival rate varies slowly, congestion peaks after the arrival rate does [81], and staffing rules need time-varying approximations [82]. These results size the capacity. However, they give no bound for an individual job once a queue has formed, and in our setting the capacity is fixed.

2.5. Relational Models for Cost Prediction

A job's cost can depend on its user, its task and the relations between them, which is the setting of relational deep learning. On the RelBench benchmarks, graph models over linked tables match or beat gradient-boosted trees fitted on the columns of a single table [83]. On single tables of about 10,000 rows, however, tree ensembles still outperform deep models [84]. We therefore compare an HGNN over users, tasks and assignments [20] with gradient-boosted trees [18] that carry engineered history features. The graph network is judged by its prediction scores only. The trees and a recurrent model are also judged by the waits that they produce in the queue.

3. Problem Model

3.1. Jobs, Servers and Dispatch

A finite set of n jobs is served by k identical servers. We say *server* for what a deployment calls an executor, a judging machine or a worker, and *job* for a submission, a function call or a test run. Job i arrives at time $a_i \geq 0$ and needs C_i units of service, which we call its *cost*, or its *work*. Service is non-preemptive. A server that starts a job keeps it until the job finishes or has run for the time limit L, at which point the system stops it. Since C_i is the work actually executed, $0 < C_i \leq L$, and a job stopped at the limit contributes exactly L. A *task* is the programming exercise that a job attempts, so many jobs share one task.

Definition 1 (Instance). *An instance consists of* $k \geq 1$ *servers, a time limit* $L > 0$*, and jobs* $1, \ldots, n$ *with arrival times* $0 \leq a_1 \leq \cdots \leq a_n$ *and costs* $0 < C_i \leq L$*. The order of the indices is the arrival order, written* $<$*.*

Ties in arrival time are broken by input order, so $j < i$ implies $a_j \leq a_i$. Phrases such as "arrived before" and "earliest-arrived" refer to this order. At a common instant, completions are processed first, then arrivals, then dispatches. Servers are work-conserving, which means that no server is idle while a job waits. A *policy* is any rule that, whenever a server is free and a job waits, picks one waiting job and starts it. The rule can be randomised or learned, and it can use priorities or predictions of any quality. We make no stochastic assumption about the arrival times, the costs or their dependence, because the load that we study follows a calendar and is not stationary. When a policy runs on an instance, it produces a schedule in the following sense.

Definition 2 (Schedule). *A schedule of an instance gives every job* i *a start time* s_i*, a server* $m_i \in \{1, \ldots, k\}$ *and a dispatch position* $\pi_i \in \mathbb{N}$ *such that (i)* $s_i \geq a_i$ *for every job* i*; (ii)* π *is injective, and* $s_j < s_i$ *implies* $\pi_j < \pi_i$*; (iii) if* $i \neq j$ *and* $m_i = m_j$*, then* $s_i + C_i \leq s_j$ *or* $s_j + C_j \leq s_i$*; and (iv) if* $a_i \leq t < s_i$ *for some job* i*, then every server* m *has a job* j *with* $m_j = m$ *and* $s_j \leq t < s_j + C_j$*. We write* $j \prec i$*, read "*j *is started before* i*", when* $\pi_j < \pi_i$*.*

Condition (iii) says that service is non-preemptive, and condition (iv) says that no server is idle while a job waits. Condition (ii) orders the jobs that start at one instant, in the order in which the policy dispatched them. A policy produces one schedule for each instance and each realisation of its randomness, and every result of Section 6 is a statement about such schedules.

Definition 3 (FCFS). *A schedule is FCFS if* $j \prec i$ *holds exactly when* $j < i$*.*

All FCFS schedules of an instance have the same start times. By induction on the arrival order, each job starts at the first instant at which a server is free, among the instants no earlier than both its arrival and the start of its predecessor. The FCFS wait of every job is therefore well defined.

3.2. What the Scheduler Knows

A job's cost is hidden when it arrives, and the scheduler learns C_i only when i completes. At arrival the scheduler sees a feature vector x_i, built from the history of earlier jobs, and a score $\hat{C}_i$ computed from x_i. This history must be causal. Write done_j for the instant at which the result of job j becomes readable, and $d \geq 0$ for the delay before it reaches the store that the features are read from. Then

$$\text{information about } j \text{ may enter } x_i \quad \Leftrightarrow \quad \mathrm{done}_j + d \leq a_i. \tag{1}$$

In a simulation driven by a log, done_j can be read on two clocks. On the *original clock* it is the time at which the result appeared on the original platform, as the log records it. In the simulated queue, the job completes at another time, which depends on the policy being simulated. A feature that is causal on the original clock can therefore use a result that the simulated queue has not yet produced. Under the *online* protocol, a job's score may use a result from the same copy of its class and term (Section 7.2) only once the simulated queue has produced that result. Results from other classes and terms keep their recorded times (Section 8.1). The prediction study of Section 4 uses the original clock. The scheduling results of Sections 8.1 and 8.2, which the abstract reports, use the online protocol. The guarantees of Section 6 apply on either clock, since they assume nothing about the scores.

https://doi.org/10.3390/math14214020

3.3. Waiting Time, the FCFS Reference and the Service Guarantee

Let s_i^P be the start time of job i in the schedule of policy P. FCFS always starts the earliest-arrived waiting job, and when several servers become free at one instant, it starts jobs in arrival order. We run FCFS on the same arrivals, costs and k as P, so that the two are compared job by job on one sample path.

Definition 4 (Excess and overtakers). *The wait of job i under P is $W_P[i] = s_i^P - a_i$, and its excess is*

$$\text{excess}_P[i] \;=\; W_P[i] - W_{\text{FCFS}}[i]. \tag{2}$$

A job j with $j > i$ and $j \prec i$ is an overtaker of i. The work that overtook i and the work that i overtook are

$$\text{In}_i \;=\; \sum_{j>i,\; j\prec i} C_j\,, \qquad \text{Out}_i \;=\; \sum_{j<i,\; i\prec j} C_j. \tag{3}$$

Definition 5 (Service guarantee). *A policy P offers a service guarantee G if $\text{excess}_P[i] \le G$ for every job i of every instance with time limit L, for every score, adversarial scores included, and for every realisation of the randomness of P.*

An operator states G in advance, and the guarantee is the proved statement that every job stays within it. The guarantee is stated on the realised sample path, so it applies to each job and not to an average, and it assumes nothing about the quality of the score. Packet-scheduling results state additive bounds of this kind for each packet, against a reference discipline [66]. Studies of batch schedulers measure the same difference after the run, against a fair start time [54]. The literature on scheduling with predictions compares with an optimal policy instead [10,13]. We take FCFS as the reference because a shared queue serves in arrival order unless a policy reorders it. The excess is then the delay that the reordering adds, and an operator can be held to account for it. Against FCFS, a budget on the work that overtakes a job bounds the excess of that job. With earliest-deadline-first or strict priority classes as the reference, a later job that ranks ahead adds delay that no such budget counts (Supplementary Section S7.7).

3.4. Metrics

The primary metric is the 99th percentile (p99) of the waiting time of the jobs that arrive in the 24 h before a deadline, written p99_{dl}. We fixed this metric before any sealed part of a log was opened. Besides it, we report the mean wait over all jobs and the largest excess, which is the quantity that a guarantee bounds. We also report the *harm*, the largest excess among the jobs that FCFS would have started within 1 s, that is, among the jobs that FCFS would not have delayed. The guard of Section 5 *fires* at a dispatch decision when some waiting job has used up its budget. It then picks the job itself, and that job can be the one that the base policy would have picked. The *firing rate* is the share of dispatch decisions at which the guard fires, with each decision weighted by the length of the queue. It equals

$$\frac{\sum_d \,|Q_d|\,\mathbf{1}\{\text{the guard fires at } d\}}{\sum_d \,|Q_d|}, \tag{4}$$

where $|Q_d|$ counts the jobs waiting just before decision d, the dispatched one included. Finally, we report the worst wait among heavy jobs, which Section 4 defines.

Improvements are expressed as the fraction of the gap closed,

$$\text{gap closed}(P) = \frac{M_{\text{FCFS}} - M_P}{M_{\text{FCFS}} - M_{\text{SJF}}}, \tag{5}$$

where M_Q is p99_{dl} under policy Q and SJF orders the waiting jobs by their true costs. A gap closed of 1 means that P matches the p99 of SJF, and 0 means that it matches FCFS. The ratio is defined only when $M_{\text{FCFS}} > M_{\text{SJF}}$, and the tables print no gap otherwise. Gap closed is not a percentage reduction in waiting time, so the tables print the absolute waits beside it. A table or figure that averages over the overlays of Section 7.2 averages this ratio and not the p99 values, unless its caption says otherwise. SJF cannot be deployed, since costs are unknown at arrival. On several non-preemptive servers, it also need not minimise our metrics. For example, its largest excess is positive, whereas that of FCFS is zero, and on one CI pool, its p99 is above that of FCFS (Section 8.7). SJF is therefore a reference and not an optimum.

3.5. Steady-State Models and This Load

Stationary models give closed forms for the mean wait of each class of a non-preemptive priority queue [85] (pp. 499–507). The SOAP (Schedule Ordered by Age-based Priority) framework analyses every policy that ranks jobs by a function of their size class and age [86]. This family contains ranking by predicted cost. Both analyses need stationary arrivals, which deadline-driven load does not supply. On a synthetic Poisson stream, the priority formula agrees with simulation to within 0.4%. On the same jobs at their recorded arrival times, however, the measured mean wait is 10.7 to 13.7 times what the formula predicts (Supplementary Section S2.5). We therefore use steady-state results as motivation only, and we state every guarantee on the realised sample path.

4. Cost Prediction

For every arriving job, the scheduler needs a number that says how long the job will run, and it must compute this number from history alone. Users and tasks form a graph, because many users attempt the same task and each user attempts many tasks. We therefore ask whether a model that reads this graph predicts better than one that reads the history of each job's own user and task. This section compares three kinds of model on the main grading log, from the CodeBench course platform (Section 7). The tree models are also fitted on a second log, from the ACcoding online judge, but the neural models are fitted on the main log only. The section ends with the choice of the model that the scheduler uses.

4.1. Targets and Measures

For ranking, the target is the conditional mean $\mathbb{E}[C_i \mid x_i]$. Among orders fixed in advance, this mean minimises the expected mean wait of a batch on one server. The reason is Smith's pairwise-interchange argument [87], applied to the expected costs (Section 5.2). The argument says nothing about the 99th percentile of the wait that we report, so Section 8.3 tests this target in the queue. The mix of jobs shifts between periods, so to compare predictors across periods we also score each model on a binary label. A job is *heavy* when its cost exceeds the 95th percentile of the training rows. On the main grading log this cut-off is 1.559 s, and heavy jobs are 1.31% of the test period against 5.00% of the training period. On the second log the cut-off is 1.314 s.

Each model therefore has two outputs, a score for the heavy label and a prediction of $\log(1 + C)$, with C in seconds. For the trees, we fit the two outputs as separate models. For the neural models, the two outputs are two heads on one shared network. For the heavy label, we report the area under the receiver operating characteristic curve (AUROC). For the predicted cost, we report the root mean squared error (RMSE) and the

Spearman rank correlation, since a ranking policy uses the order of the scores and not their level.

4.2. Protocol

Every model is trained on the same rows and tested on the same term, under the same visibility rule (1). The neural models also use the term 2022-1 to choose their learning rate and weight decay and to stop training, and the trees do not use it (Supplementary Section S1.4). Historical features are computed by streaming the log in the order in which results became available, and by reading each user's and each task's state at the query instant. On 3000 randomly drawn development jobs, recomputing these features from scratch gives 0 disagreeing rows in all eight configurations that we tested. For each test period, we use a model that was fitted only on earlier periods and frozen before the first arrival of the period. The heavy cut-off inside the history features is refitted on that model's own training rows. The heavy label on which we evaluate keeps one cut-off, the 95th percentile of executed cost in the terms 2018-1 to 2019-2, 1.559 s. When results reach the features ten minutes late, the full tree model M4 described below loses part of its AUROC and half of its average precision on the heavy label. Supplementary Section S1 gives these audits.

Table 2 lists the three kinds of model and the information that each one uses. Every model in the table is trained on the terms 2018-1 to 2019-2 and tested on 2022-2, and its results are averaged over 3 seeds. A term is written as year and semester. The scheduler uses a different fit on the features of M4: the expected-cost regressor of Section 5.2, which is refitted on rolling forward periods (Supplementary Section S3.9).

- Gradient-boosted trees [18] over five feature groups, namely the task's history (M1), the user's history (M2), static features of the submitted code (M3), all of these together (M4) and M4 with one-hop aggregates over related users and tasks (M5).
- A gated recurrent unit (GRU) [19] over the user's earlier jobs, used alone (R1) and as a state vector added to the features of M4 inside the trees (R1S).
- A heterogeneous graph neural network (HGNN) of the R-GCN type [20], over users, tasks, the assignments that group tasks and the class cohorts that group users. It is tested as a ladder of five models. The first is a multilayer perceptron on the features of M4 (N0). The other four use the graph architecture. G0U has every neighbour set empty and no group nodes, G0 has the group nodes but no edges, G1 has the real edges, and G2 has the edges shuffled within each period. G2 keeps each node's number of neighbours and changes only who they are. So if the edges carry information, G2 must lose it.

Table 2. Predictors on the main grading log (CodeBench), 40,844 rows of one test term (2022-2). Each entry is the mean over seeds 3, 4 and 5, with the standard deviation over seeds in parentheses. AUROC is computed on the output for the heavy label, and the Spearman correlation on the predicted $\log(1 + C)$. The second grading log and the remaining measures are in Supplementary Table S2.

Kind	Model	Information	AUROC	Spearman
Trees	M1	task history	0.7960 (0.0149)	0.219 (0.0142)
	M2	user history	0.8591 (0.0046)	0.302 (0.0013)
	M3	static features	0.8632 (0.0025)	0.290 (0.0024)
	M4	all of the above	0.9283 (0.0020)	0.312 (0.0065)
	M5	M4 and one-hop aggregates	0.9302 (0.0002)	0.344 (0.0019)
GRU	R1	user's earlier jobs	0.9239 (0.0072)	0.313 (0.0049)
	R1S	M4 and the GRU state	0.9366 (0.0011)	0.329 (0.0096)
HGNN	N0	M4, no graph	0.9209 (0.0083)	0.278 (0.0178)

G0U	graph model, no edges or group nodes	0.9348 (0.0008)	0.321 (0.0070)
G0	graph model, no edges	0.9325 (0.0011)	0.324 (0.0092)
G1	graph model, real edges	0.9324 (0.0023)	0.329 (0.0023)
G2	graph model, shuffled edges	0.9305 (0.0006)	0.323 (0.0096)

4.3. What the Comparison Shows

On the one test term, the graph's edges add no gain that the intervals can detect. The intervals also bound the size of any such gain. One-hop aggregates change the AUROC of the trees by $+0.0019$ on the main log and by -0.0002 on the second (M5 − M4). In the HGNN ladder, removing every edge (G0) or shuffling them (G2) leaves the AUROC within 0.002 of G1. The paired interval for G1 − G0, -0.0001 $[-0.0019, 0.0018]$, contains zero. The one step in the ladder that matters is N0 to G0U, $+0.0139$ $[+0.0090, +0.0195]$ of AUROC. This step changes the encoder while the edges stay empty. The intervals resample test users and hold each fitted model fixed. Across the three seeds, the AUROC of N0 has a standard deviation of 0.0083 (Table 2), so this step is less than two such deviations.

The neural models differ from the trees on tasks with little history of their own, and the edges do not cause the difference. On the 1092 test rows whose task has fewer than five visible earlier results, G1 beats M4 by $+0.1464$ $[+0.0997, +0.1887]$ in Spearman correlation. Models without a graph give most of this gain. The perceptron N0 uses only the features of M4, yet it beats M4 by $+0.1064$. The edgeless G0U beats M4 by $+0.1419$ $[+0.0994, +0.1803]$. Adding the real edges to G0U changes the correlation by $+0.0045$ $[-0.0163, +0.0248]$. G1 beats its shuffled control G2 on these rows by $+0.0282$, because shuffled edges make the model worse than no edges (G2 − G0U is -0.0238 $[-0.0453, -0.0025]$). The intervals above resample test users. The 1092 rows come from 101 users, 219 tasks and 58 assignments. When tasks, assignments, or users and tasks together are resampled instead, the widest interval for G0U − M4 is $[+0.0649, +0.2195]$, and the widest for G1 − G0U is $[-0.0262, +0.0388]$.

Prediction scores and queueing outcomes rank the models differently. R1S has the highest AUROC, 0.9366, which exceeds that of M4 by $+0.0083$ $[+0.0038, +0.0131]$. In the queue, with the same ranking objective as M4, R1S gains $+0.007$ $[0.003, 0.011]$ of the gap at utilisation 0.8. At 0.5 it changes the gap closed by -0.010 $[-0.022, -0.002]$, and at 1.0 the difference is unresolved (Section 7.2 defines the three loads). These differences are paired within the earlier study that fitted R1S, against that study's own run of shortest predicted job first on the expected-cost score (SPJF-E, Section 5.2; Supplementary Table S16). M5 has the higher AUROC of the two full tree models, but it loses in the queue at all three loads, by 0.012, 0.008 and 0.010 of the gap. Each of these intervals excludes zero. We therefore chose M4 for the scheduler on its queueing outcome and not on prediction scores. These comparisons ran on the primary development pool, which also gives the development figures of Section 8, so the choice is not held out. The sealed terms of Section 8.2 test it. Supplementary Table S16 lists R1S in the queue, and Supplementary Section S5 lists the archived analyses that compare M5.

The history that matters differs between platforms. The user's history leads on the main log, and the task's history leads on the second by 0.11 of AUROC (Supplementary Table S2). M4 carries both groups and has a higher AUROC than any single group on both logs. On these two platforms, the scheduler's features therefore need not be chosen per platform. Supplementary Section S1.2 compares the feature groups on the four logs from outside education.

Prediction is cheap next to the queue. The median job on the main log runs for 204 ms. When we charge each model's measured cost per job to the predictive policies, the gap closed changes by less than 0.001 at every load (Supplementary Section S1).

5. Scheduling Algorithm

5.1. Overview of the Method

The method has three parts, and Figure 2 shows how they act on one job. When job i arrives, we compute its features x_i from earlier results that are visible under (1). A regressor maps x_i to the score $\hat{C}_i$ (Sections 4 and 5.2), and we fit this regressor only on earlier periods. The job then joins the waiting queue, and its score does not change while it waits.

Whenever a server is free and a job waits, the guard makes the dispatch decision. It first looks for a waiting job whose overtake account, the work of later arrivals that have already completed, has reached its budget. If such a job exists, the guard starts the earliest-arrived one. Otherwise, it lets the base policy start the waiting job with the smallest score. When a job completes, its cost becomes known. The guard then adds this cost to the account of every earlier-arrived job that is still waiting. The job's result also enters the history from which later features are computed.

Each part has its own role and its own inputs. The scores determine which jobs go first, so they produce the reduction in waiting time. The guard limits how far any job can fall behind its FCFS position. Its bound is valid for every score, a wrong one included, because the accounts are charged with the costs of executed jobs and never with predictions (Theorem 3). The operator states the bound G, which fixes the cap $B_{\max}$ without data (Section 5.4). We choose the budget shape on validation data and evaluate the frozen method once on sealed test terms (Section 7.3).

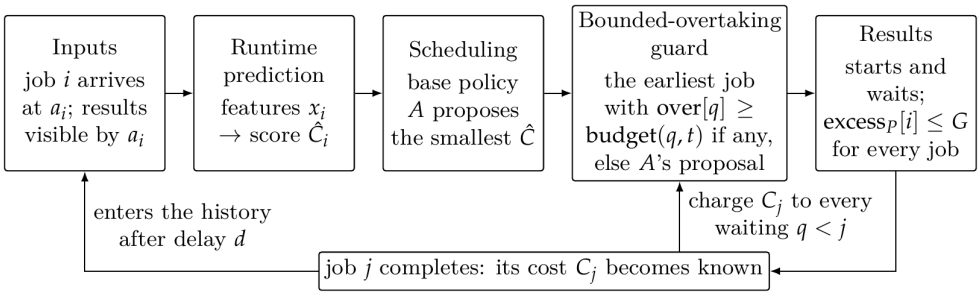


Figure 2. The method on one job. The prediction stage turns the visible history into a score at arrival. The base policy proposes a job by score, and the guard replaces the proposal when a waiting job has been overtaken by its budget of work. A completion reveals the job's cost, which is charged to the earlier-arrived waiting jobs and enters the history used for later predictions. The guarantee excess$_P[i] \leq G$ rests only on the guard and these charges (Theorem 3), not on the scores.

5.2. Ranking by Expected Cost

When a server becomes free, the base policy starts the waiting job with the smallest score. For a batch on one server, ordering the jobs by their conditional mean costs minimises the expected mean wait among orders fixed in advance. This follows from Smith's pairwise-interchange argument [87], applied to the expected costs by linearity of expectation. The conclusion fails on $k \geq 2$ servers. For example, on two servers, let job u cost 2 and job w cost 3, and let job v cost 0.1 with probability 0.9 and 100 with probability 0.1. Ordering by conditional mean starts u and w and gives an expected mean wait of 2/3, whereas starting u and v gives 0.096667. We therefore treat expected-cost ordering as an empirically motivated base policy, not as a claim of optimality.

Squared error on raw seconds and the Tweedie deviance both have the conditional mean as their population optimum. However, their fitted orders can still differ, because of finite model capacity, the link function and regularisation. We therefore compare the objectives empirically, with the features and model of Section 4 held fixed. The expected-cost policy is shortest predicted job first on the expected-cost score (SPJF-E). We fit its score on raw seconds with a Tweedie loss of power 1.5, which accommodates mass near zero and a long right tail [88,89]. The power was set in advance. Every scheduling result uses one fit of this model for each target term, with seed 3 (Supplementary Section S1.4). The log-scale policy is shortest predicted job first on the log-scale score (SPJF-log). For it, we fit the same pipeline by squared error on $\log(1 + C)$. By Jensen's inequality, the back-transform of its population optimum, $\exp(\mathbb{E}[\log(1 + C) \mid x]) - 1$, is no larger than the conditional mean $\mathbb{E}[C \mid x]$.

We use only the order of a score. Its level does not transfer between periods, because the share of heavy jobs differs between training and test periods by a factor of several. The scheduler therefore never reads a score as a time or a threshold. Section 8.3 compares these and other objectives in the queue. However, no choice of score bounds the worst case. Whatever the score, it ranks some jobs late, because they are long or because their costs are overestimated. Later arrivals that the score ranks ahead of such a job can then pass it again and again. The guard below bounds this, and so does a maximum waiting time (Section 6.6).

5.3. The Overtake-Budget Guard

Fix a base policy A. For every waiting job q and every time t let

$$\text{over}[q](t) \;=\; \sum\{\, C_j \,:\, j > q,\ j \text{ has completed by } t \,\}, \tag{6}$$

the total cost of the jobs that arrived after q and have finished by t. Such a job ran while q waited, so it overtook q. Each waiting job has a budget $\text{budget}(q, t)$ on this quantity, and the job is *eligible* when its account has reached its budget, $\text{over}[q](t) \geq \text{budget}(q, t)$. Whenever a server is free and the queue is not empty, the guard starts the earliest-arrived eligible job, if there is one. We then say that the guard *fires*. Otherwise, it lets A choose (Algorithm 1 and Figure 3). Serving the earliest-arrived eligible job matters when budgets differ between jobs. This choice ensures that whenever a job j starts ahead of an earlier-arrived waiting job i, the job i was not eligible at that instant. The per-job bound of Section 6 rests on this step. With a constant budget B, the account is non-increasing in arrival order. The rule then reduces to a test on the head of the queue, and $B = 0$ gives FCFS.

The guarantee depends on the budget only through an upper bound on it. If $0 \leq \text{budget}(q, t) \leq B_{\max}$ for every waiting job at every instant, then every job waits at most $W_{\text{FCFS}} + B_{\max}/k + (3 - 2/k)L$, whatever the base policy, the arrivals and the predictions (Theorem 3). Below the cap, the budget determines how much of the base policy's benefit survives and which jobs bear its cost. We study the family

$$\text{budget}(q, t) \;=\; \min\big(\, B_0 + \gamma\, n_q + \eta\, k\, (t - a_q),\ B_{\max} \,\big), \tag{7}$$

with $B_0, \gamma \geq 0$, $0 \leq \eta < 1$ and n_q the number of jobs waiting when q arrives. We study three shapes in this family, namely *constant* ($\gamma = \eta = 0$), *age-relative* ($\eta > 0$) and *queue-length* ($\gamma > 0$). In the age-relative shape, η is the share of the system's capacity during a job's wait that can go to jobs that arrived after it. The allowance therefore grows with the wait that the job has already had. The term γn_q widens the allowance only for jobs that arrive into a long queue. All of B_0, γ and $B_{\max}$ are measured in work and scale with k.

Algorithm 1. Overtake-budget guard wrapped around a base policy A

parameters: cap $B_{\max} \geq 0$; shape B_0, $\gamma \geq 0$, $\eta \in [0, 1)$; base policy A
state: waiting set $Q \leftarrow \emptyset$; over$[q] \leftarrow 0$ for every job q
while some job has not started **do**
    $t \leftarrow$ next event time
    for all jobs c completing at t **do** ▷ C_c becomes known here and nowhere earlier
        for all $q \in Q$ with $q < c$ **do**
            over$[q] \leftarrow$ over$[q] + C_c$
        end for
    end for
    for all jobs i arriving at t, in the order $<$ **do**
        $n_i \leftarrow |Q|$; $Q \leftarrow Q \cup \{i\}$; over$[i] \leftarrow 0$
    end for
    while a server is free **and** $Q \neq \emptyset$ **do** ▷ dispatch decision
        $E \leftarrow \{q \in Q : \text{over}[q] \geq \min(B_0 + \gamma n_q + \eta k(t - a_q), B_{\max})\}$
        if $E \neq \emptyset$ **then**
            $j \leftarrow$ the earliest-arrived member of E ▷ guard fires
        else
            $j \leftarrow A(Q, t)$ ▷ base policy decides
        end if
        start j on the free server; $W[j] \leftarrow t - a_j$; $Q \leftarrow Q \setminus \{j\}$
    end while
end while

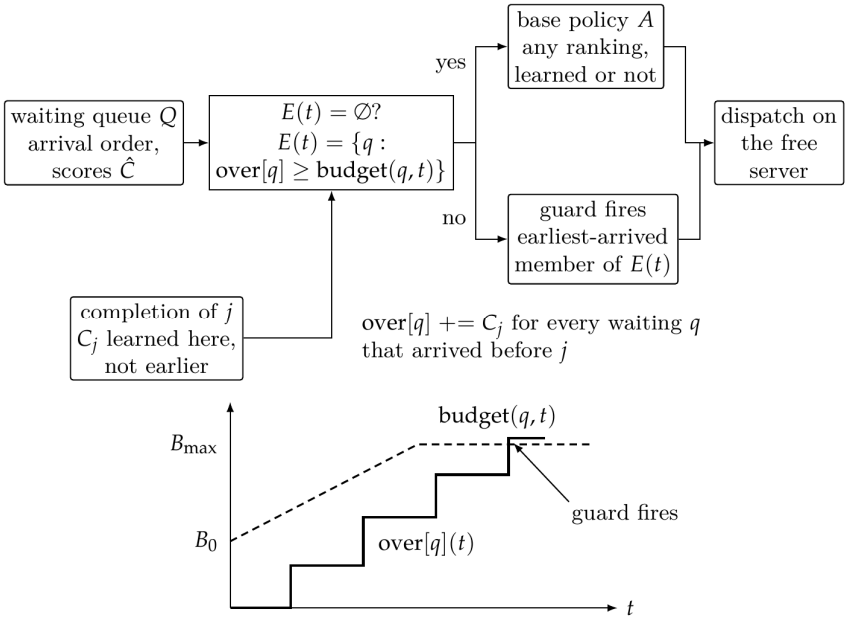


Figure 3. The guard of Algorithm 1 wrapped around a base policy (above), and the account it keeps for one waiting job q (below). Only completed work is charged, because a job's cost is unknown until it finishes. The account over$[q]$ therefore rises by one step at each completion of a job that arrived after q. When the account reaches the budget, q becomes eligible, and the guard starts the earliest-arrived eligible job, which need not be q. The budget drawn is the age-relative shape, rising from B_0 at rate ηk and capped at $B_{\max}$. The bound depends on the cap alone, so any budget below the dashed ceiling carries the same guarantee. The lower panel is a schematic.

5.4. Setting the Guard

An operator states a bound G, the number of seconds by which any job may exceed its FCFS wait. The operator then sets $B_{\max} = k(G - (3 - 2/k)L)$ (Corollary 3). No data are needed for this step. A bound of at most $(3 - 2/k)L$ can be guaranteed this way only with $B_{\max} = 0$, which is FCFS. The shape parameters B_0, γ and η do not affect the bound, and we choose them on validation data by a fixed rule (Section 7.3). At $G = 300$ s the rule selects the age-relative shape with $B_0 = 15k$ s of work and $\eta = 0.50$. At $G = 600$ s it selects the same shape with $B_0 = 30k$ s and $\eta = 0.75$. At $G = 1200$ s it selects the queue-length shape with $B_0 = 30k$ s and $\gamma = 4k$ s (Supplementary Table S5).

Two rules need no grid search. The first is the constant budget $B_0 = B_{\max}/2$. Since the budget never exceeds B_0, Theorem 3 with B_0 in place of $B_{\max}$ bounds every excess by $(G + (3 - 2/k)L)/2$, halfway between the floor and G. The second is the age-relative shape with the parameters selected for the main tables. These parameters carry over to other numbers of servers. Section 8.6 compares both with the other shapes on one to three servers.

The guard charges the executed cost of an overtaker when it completes, because a job's cost is known only then. A job still running contributes nothing yet. Consider the moment when the last overtaker of a job i starts. Up to $k - 1$ earlier overtakers can still be running then, and the starting overtaker is not charged either. The work that actually passes i can therefore exceed the budget by at most kL. The additive term of the guarantee covers this excess. For this reason, the guard needs no prediction to enforce its bound. Our implementation keeps a Fenwick tree [90] over completed work, indexed by arrival rank over all n jobs of a trace, and a segment tree over a sliding window of $R = 2^{22}$ ranks. Each query or update then costs $O(\log n)$ or $O(\log R)$, independent of the number of waiting jobs. Trees indexed by the waiting jobs alone would cost $O(\log|Q|)$. All budget arithmetic is done in integer microseconds, so the eligibility test is exact (Supplementary Section S5). Beyond the base policy, the guard stores two 8-byte integers for each job of the trace and $3R$ 8-byte entries for the segment tree. For the 17,634,760 jobs of one overlay, a trace that Section 7.2 builds from the main log, this is about 383 MB. The window restarts at the next arrival whenever the queue empties. It is rebuilt only when the waiting ranks span more than R. On the first overlay, the widest span was 36,547 ranks, and no rebuild occurred. Section 8.10 measures the cost.

5.5. Compared Policies

Table 3 names the policies compared in Section 8. Our method is Guard(G), the guard with bound G wrapped around SPJF-E. The other rows separate its parts, test it or stand for existing wrappers.

Table 3. Policies compared in the experiments.

Name	Role
FCFS	Arrival-order baseline and the reference of the guarantee
SJF	Orders by true cost, cannot be deployed, and is the reference of the gap closed
SPJF-E	Orders by the expected-cost score of Section 5.2, unguarded
SPJF-log	Orders by the log-scale score, unguarded
Guard(G)	SPJF-E inside the guard with bound G and the selected shape (the method)
Guard-fixed(G), Guard-age(G), Guard-queue(G)	The same guard with the best constant, age-relative or queue-length shape

Fixed(G)	The same guard with the whole cap as budget, budget $\equiv B_{\max}$
Skip(G)	A guard that counts overtakes instead of work, charged at start, at the count that gives bound G
Timeout(θ)	A maximum waiting time, which serves the earliest-arrived job once it has waited θ (Section 6.6)
Aging	Orders by $\hat{C}_i - \omega(t - a_i)$, the score minus a credit ω per second of wait, with no proved bound
Block(b)	Orders by score inside consecutive blocks of b arrivals and serves the earliest block that has a waiting job, a stand-in for window ordering by predicted cost [73]
CBF-L, CBF-P	Conservative backfilling [56], planned on the limit L or on the SPJF-E predictions
Slack-L(G), Slack-P(G)	Slack-based backfilling [58] with slack G, planned on L or on the SPJF-E predictions
Corrupted scores	SPJF-E and Guard(G) under a reversed, a random and an inverted-tail score, which ranks the longest 1% of jobs as the shortest

Every simulation that includes a guard checks the per-job bound of Section 6 on every job while it runs (Section 8.5).

6. Theory

Every result in this section is stated for one fixed run, that is, for one arrival sequence, one vector of costs and one realisation of the randomness, if any, of the base policy. No stationarity, no independence and no assumption about the scores enters a proof, and the upper bounds apply even to a base policy that knows the future. An identity ties a job's delay relative to FCFS to the work that overtook it (Sections 6.2–6.4). Two bounds turn it into the guarantee of Algorithm 1 (Section 6.5), and three further results cover a maximum waiting time, an absolute bound and completion reports that arrive late (Sections 6.6–6.8). Full proofs and the extremal constructions are in Supplementary Sections S4 and S8.

We also proved the results of this section in Lean 4 [91] with the Mathlib library [92]. The Lean model is that of Definitions 1 to 6, the last of which is stated in Section 6.5, and a policy enters it only as a condition on a schedule. Every proof depends only on the three standard axioms of Lean. Each lemma, theorem, corollary and proposition below has a Lean counterpart, with three restrictions. For Proposition 3 the Lean files prove the bounds but not their attainment. For Proposition 4 they prove the first sentence and the zero-budget case of the second. For Theorem 4 they prove the inequality but not the order of its finite-sample term. The Lean theorems take the existence of an FCFS run, a guard run or an SJF run on a general input as a hypothesis. Some results are proved only on paper: the extremal constructions, the agreement of replicas, the refund variant of the charge and the additional results of Supplementary Section S7. Supplementary Section S12 maps each result to its Lean theorems, and the proof files are in the repository named in the Data Availability Statement.

6.1. Conventions

We use the model of Section 3, and Table 4 collects the symbols of this section with the place where each is defined. The position of a job in the arrival order is its *rank*, so $j < i$ means that j has the lower rank. Three readings fix what the proofs count. First, several jobs can start at the same instant. We call them a *phase* and let the policy start them one after another, so that "j is started before i", written $j \prec i$, is defined even inside a phase.

Within a phase, FCFS starts the jobs in arrival order. Second, consider a job of higher rank that is started in the same phase as i but ahead of it. This job overtakes i and contributes its whole C_j to In_i, although it executes nothing while i waits. Third, the remaining work at the instant a_i is read after the jobs arriving at a_i have been added.

Table 4. Principal notation of Section 6. Symbols used in a single result are defined there.

Symbol	Meaning	Defined in
$k,\ L$	number of servers; time limit, so every cost satisfies $0 < C_i \le L$	Section 3
$a_i,\ C_i$	arrival time and cost (executed work) of job i	Section 3
$j \prec i,\ j \prec i$	j arrives before i (lower rank); j is started before i	Sections 3 and 6.1
$s_i^P,\ W_P[i]$	start time and wait $s_i^P - a_i$ of job i under policy P	Section 3
$\mathrm{excess}_P[i]$	$W_P[i] - W_{\mathrm{FCFS}}[i]$	Equation (2)
In_i	total cost of the higher-rank jobs started before i (the work that overtook i)	Equation (3)
Out_i	total cost of the lower-rank jobs started after i (the work that i overtook)	Equation (3)
G	service guarantee: $\mathrm{excess}_P[i] \le G$ for every job	Section 3
$U_Q(t)$	unfinished work at time t under policy Q	Section 6.2
$R_i^Q,\ \beta_i^Q$	remaining work at a_i of the lower-rank jobs; remaining work at s_i^Q of the jobs started before i and still running	Section 6.3
$\mathrm{over}[q](t)$	total cost of the higher-rank jobs completed by time t	Equation (6)
$\mathrm{budget}(q,t),\ B_{\max}$	the budget of the guard for job q at time t, and its cap	Section 5.3
$B_0,\ \eta,\ \gamma$	parameters of the age-relative and queue-length budget shapes	Section 5.3
θ	waiting time after which Timeout(θ) starts the earliest-arrived job	Section 6.6
δ	largest delay of a completion report	Section 6.8
$F_i,\ F^*$	flow time $W[i] + C_i$; the smallest achievable largest flow time	Section 6.5

6.2. The Workload Gap Between Two Policies

Write $U_Q(t)$ for the unfinished work at time t under policy Q, counting every job with $a_j \le t$.

Lemma 1 (Workload comparison). *Let Q_1 and Q_2 be non-preemptive work-conserving k-server policies run on the same input. Then $|U_{Q_1}(t) - U_{Q_2}(t)| \le (k-1)L$ for every t. For $k \ge 2$ the inequality is strict, $|U_{Q_1}(t) - U_{Q_2}(t)| < (k-1)L$, and the constant is the exact supremum, approached but never attained. At $k = 1$ both sides vanish, because one server executes work at rate 1 whenever at least one job is present, so every work-conserving policy has the same unfinished work at every instant.*

Proof of the inequality $|U_{Q_1}(t) - U_{Q_2}(t)| \le (k-1)L$. Let $N_Q(t)$ be the number of jobs present under Q and put $D = U_{Q_1} - U_{Q_2}$. An arrival adds the same cost to both unfinished works, so D has no jumps. Between arrivals, a work-conserving policy with N jobs present keeps $\min(N,k)$ servers busy, so U_Q falls at rate $\min(N_Q,k)$ and D changes at rate $\min(N_{Q_2},k) - \min(N_{Q_1},k)$.

Suppose $U_{Q_1}(u) > (k-1)L$ at some instant u. Every job carries at most L, so more than $k-1$ jobs are present under Q_1, all k of its servers are busy, and the rate of D at u is $\min(N_{Q_2},k) - k \le 0$: D does not increase while U_{Q_1} stays above $(k-1)L$. Now fix t and let t_0 be the last instant up to t at which $U_{Q_1} \le (k-1)L$, taken as a limit from the left because U_{Q_1} jumps up at arrivals. Such instants exist, since $U_{Q_1} = 0$ before the first arrival. At t_0, $D \le U_{Q_1} \le (k-1)L$ because $U_{Q_2} \ge 0$, and on $(t_0,t]$ the difference D does not increase, so $D(t) \le (k-1)L$. Exchanging Q_1 and Q_2 gives $D(t) \ge -(k-1)L$. Supplementary Section S4.1 proves the strict inequality for $k \ge 2$, and Section S8.1 gives the construction showing that $(k-1)L$ cannot be lowered. □

A similar argument compares multiserver shortest remaining processing time (SRPT) [93] with a single server k times as fast. There, with preemption, it bounds the difference in the work of the jobs smaller than a cutoff x by kx [94]. Here, we use it to compare any two non-preemptive work-conserving policies over all of the work. FCFS also minimises the unfinished-work vector in a stronger order, but only among policies whose assignment of jobs to servers does not depend on their costs [95]. The policies compared here, however, do not have this property.

6.3. The Net-Overtake Identity

For a job i and a policy Q, let R_i^Q be the remaining work at time a_i, under Q, of the jobs that arrived before i, and let β_i^Q be the remaining work at s_i^Q of the jobs started before i that are still running then. Write

$$\Delta_i \;=\; k(W_P[i] - W_{\mathrm{FCFS}}[i]) \;-\; (\mathrm{In}_i - \mathrm{Out}_i). \tag{8}$$

Theorem 1 (Net-overtake identity). *For every non-preemptive work-conserving k-server policy P, every input and every job i,*

$$\Delta_i \;=\; \left(R_i^P - R_i^{\mathrm{FCFS}}\right) - \beta_i^P + \beta_i^{\mathrm{FCFS}}, \qquad |\Delta_i| \;\le\; 2(k-1)L, \tag{9}$$

with strict inequality for $k \ge 2$. Equivalently, $|\,W_P[i] - W_{\mathrm{FCFS}}[i] - (\mathrm{In}_i - \mathrm{Out}_i)/k\,| \le (2 - 2/k)L$. At $k = 1$ the error term vanishes and $W_P[i] - W_{\mathrm{FCFS}}[i] = \mathrm{In}_i - \mathrm{Out}_i$ exactly.

The core of the identity is the sample-path balance of work that underlies conservation laws for queues [96–99]. Summed over all jobs, the identity gives back Kleinrock's law (Supplementary Section S7.3). What it adds is the split of one job's extra wait by rank: the work that overtook the job and the work that it overtook, with an error of at most $(2 - 2/k)L$. The guard of Section 5 limits the first of these terms job by job.

Proof of Theorem 1. *Work done while i waits.* Job i waits throughout $[a_i, s_i^P)$, so by work conservation every server is busy on that interval, and exactly $k\,W_P[i]$ units of work are executed there, none of it on i. We split this work by rank.

A job j of higher rank arrives at or after a_i, so it runs inside the interval only if it is dispatched before i, that is, only if $j \prec i$. The work executed on these jobs is therefore $\mathrm{In}_i - \beta^{\mathrm{new}}$, where β^{new} is the work that the overtakers still running at s_i^P have left. An overtaker dispatched at s_i^P itself adds its full cost to both terms and nothing to the executed work. The jobs of lower rank are all present at a_i and hold R_i^P there. At s_i^P they still hold Out_i, the full cost of those not yet started, plus β^{old}, the remainder of those started before i and still running. The work executed on them is therefore $R_i^P - \mathrm{Out}_i - \beta^{\mathrm{old}}$. Adding the two parts, with $\beta_i^P = \beta^{\mathrm{old}} + \beta^{\mathrm{new}}$,

$$k\,W_P[i] \;=\; R_i^P + \mathrm{In}_i - \mathrm{Out}_i - \beta_i^P. \tag{10}$$

Size of β. The jobs counted in β_i^P run on distinct servers, none of them the server that takes i, so there are at most $k-1$ of them, each with at most L left: $0 \le \beta_i^P \le (k-1)L$. *The reference.* Under FCFS no job of higher rank starts while i waits, and every job of lower rank starts no later than i, so $\mathrm{In}_i = \mathrm{Out}_i = 0$ and (10) reads $k\,W_{\mathrm{FCFS}}[i] = R_i^{\mathrm{FCFS}} - \beta_i^{\mathrm{FCFS}}$, with $0 \le \beta_i^{\mathrm{FCFS}} \le (k-1)L$.

Comparison. At a_i the two systems hold the same jobs, and the jobs of rank i or higher are untouched in both, so $R_i^P - R_i^{\mathrm{FCFS}} = U_P(a_i) - U_{\mathrm{FCFS}}(a_i)$, which Lemma 1 places in $[-(k-1)L, (k-1)L]$. Subtracting the FCFS instance of (10) from the instance for P gives the identity (9). Its three terms lie in known ranges: $R_i^P - R_i^{\mathrm{FCFS}}$ in $[-(k-1)L, (k-1)L]$ and β_i^P, β_i^{FCFS} each in $[0, (k-1)L]$. The largest value of Δ_i takes the first at its top,

β_i^{FCFS} at its top and β_i^P at 0, which is $2(k-1)L$. The smallest is the mirror image, $-2(k-1)L$. At $k=1$ all three ranges are $\{0\}$. The strict inequality for $k \geq 2$ comes from the strict form of Lemma 1 (Supplementary Section S4.1). □

Corollary 1 (A work budget is sufficient). *If a policy keeps $In_i - Out_i \leq Z$ for every job, in particular if it keeps $In_i \leq Z$, then $excess_P[i] \leq Z/k + (2-2/k)L$ for every job.*

Proof of Corollary 1. By Theorem 1, $k\,\mathrm{excess}_P[i] \leq \mathrm{In}_i - \mathrm{Out}_i + 2(k-1)L \leq Z + 2(k-1)L$. Dividing by k gives the claim. □

Suppose that a scheduler never lets more than Z units of work pass a waiting job. Whatever it ranks by, it then keeps every wait within $Z/k + (2-2/k)L$ of the FCFS wait. The constant cannot be improved.

Theorem 2 (Tightness of Theorem 1). *For every fixed $k \geq 2$, time limit $L > 0$ and $\varepsilon > 0$, there are inputs, work-conserving policies P and jobs i with $\Delta_i > 2(k-1)L - \varepsilon$, and there are others with $\Delta_i < -2(k-1)L + \varepsilon$. Hence, $sup|\Delta_i| = 2(k-1)L$, a supremum that is never attained.*

Proof sketch of Theorem 2. Both witnesses start from a cascade. In each of m rounds, $k-1$ jobs of cost L arrive followed by many short jobs, and P runs the short jobs first. After m rounds, FCFS has finished everything while P still holds $k-1$ long jobs in service, so $U_P - U_{\mathrm{FCFS}}$ reaches $(k-1)L\,(1-((k-1)/k)^m)$. A victim is then placed so that FCFS starts it at once while P holds it back (upward), or the mirror image (downward). The instances use hundreds of jobs and are rescaled to keep L fixed. Supplementary Section S8.2 gives both in full. Non-attainment follows from Lemma 1. □

In the downward direction, half of the extremal value comes from the convention of Section 6.1 that charges a same-phase overtaker in full. Supplementary Section S4.3 gives the variant that charges executed work only.

6.4. A Work Budget Is Also Necessary

A policy that guarantees anything about individual jobs is forced to bound the work that overtakes them.

Lemma 2 (Outbound work under a guarantee). *Let i be a job, and suppose $excess_P[i] \leq G$ and $excess_P[j] \leq G$ for every job j counted in Out_i. Then $Out_i \leq k\,(G - excess_P[i] + L)$.*

Proof of Lemma 2. A job j counted in Out_i has lower rank than i, so FCFS starts it no later than i: $s_j^{\mathrm{FCFS}} \leq s_i^{\mathrm{FCFS}}$. Its own guarantee gives $s_j^P \leq s_j^{\mathrm{FCFS}} + G \leq s_i^{\mathrm{FCFS}} + G = s_i^P - \mathrm{excess}_P[i] + G$, and it starts after i, so $s_j^P \geq s_i^P$. Every such job therefore starts inside the window $[s_i^P,\, s_i^P + w]$ of length $w = G - \mathrm{excess}_P[i] \geq 0$. Of the jobs that one server starts inside this window, each except the last finishes before the next one starts, so their costs add up to at most w, and the last adds at most L. Summing over the k servers gives $\mathrm{Out}_i \leq k(w+L)$. □

Corollary 2 (A work budget is necessary). *If P satisfies $excess_P[j] \leq G$ for every job on some input, then $In_i \leq kG + (3k-2)L$ for every job i.*

Proof of Corollary 2. Theorem 1 gives $\mathrm{In}_i \leq k\,\mathrm{excess}_P[i] + \mathrm{Out}_i + 2(k-1)L$. The guarantee holds for every job, so Lemma 2 applies and

$$\mathrm{In}_i \;\le\; k\,\mathrm{excess}_P[i] + k(G - \mathrm{excess}_P[i] + L) + 2(k-1)L \;=\; kG + (3k-2)L.$$

□

The two corollaries say that bounding In_i and offering a per-job guarantee are the same requirement, up to an additive $O(kL)$. A bound on the number of overtakes, as in the mechanisms of Section 2, is not necessary for a guarantee. It gives a guarantee only if each pass is charged the worst case L. For a given job, that guarantee is then looser than the one from a work budget, by the ratio of L to the mean cost of the jobs that overtake the job. Against the mean cost of all jobs on the main grading log, the nominal ratio is 150 ($L = 60$ s against 0.40 s). Supplementary Section S4.4 proves both halves and gives the conversion from a count to a bound.

6.5. The Guarantee of the Guard

Nothing below restricts the base policy, which can be randomised, learned or clairvoyant. Nothing restricts the budget rule beyond the cap either. The budget can depend on the job, decrease over time, and take different values at two decisions made at the same instant. The proofs use the guard only through the following property of its schedules.

Definition 6 (Guard property). *A job q is waiting at the start of job j if $a_q \le s_j$ and either $q = j$ or $j \prec q$. It is eligible there if, in addition, $\mathrm{over}[q](s_j) \ge \mathrm{budget}(q, s_j)$, where $\mathrm{over}[q](s_j)$ counts the completions at times up to s_j and the budget is the value in force at the decision that starts j. A schedule has the guard property if, whenever some job is eligible at the start of j, the job j is the earliest-arrived eligible job.*

Algorithm 1 processes completions before dispatches and starts the earliest-arrived eligible job whenever there is one, so every schedule that it produces has the guard property.

Theorem 3 (Guard bound). *Fix a cap $B_{max} \ge 0$ and a budget rule that respects the cap, that is, $0 \le budget(q,t) \le B_{max}$ for every waiting job q and every dispatch decision at time t. For every input, let P be any schedule of it (Definition 2) that has the guard property for that rule. Every run of Algorithm 1 with that rule, around any base policy, is such a schedule. For every job i, writing $W = W_P[i]$ for the wait under P,*

$$\mathrm{In}_i \;<\; B_{\max} + kL, \tag{11}$$

$$W \;<\; W_{\mathrm{FCFS}}[i] + B_{\max}/k + (3 - 2/k)L, \tag{12}$$

where (11) is asserted whenever i has at least one overtaker.

Proof of Theorem 3. Suppose i has an overtaker, let j be the overtaker started last, and let t be its start. Job i is waiting when j is started, since i starts after j, possibly in the same phase. The guard starts the earliest-arrived eligible job whenever there is one, and i arrived before j, so i was not eligible at that decision: $\mathrm{over}[i](t) < \mathrm{budget}(i,t) \le B_{\max}$. The overtakers of i started before j are of two kinds. Those completed by t have total cost $\mathrm{over}[i](t)$. Those still running at t occupy distinct servers other than the one taking j, so there are at most $k-1$ of them, each of cost at most L. With j itself, of cost at most L,

$$\mathrm{In}_i \;<\; B_{\max} + (k-1)L + L \;=\; B_{\max} + kL,$$

which is (11). For (12), Theorem 1 with $\mathrm{Out}_i \ge 0$ gives $k\,(W - W_{\mathrm{FCFS}}[i]) \le \mathrm{In}_i + 2(k-1)L < B_{\max} + kL + 2(k-1)L = B_{\max} + (3k-2)L$, and dividing by k gives (12). If i has

no overtaker, $\mathrm{In}_i = 0$ and the same computation gives an excess of at most $(2 - 2/k)L$, which is below the bound because $L > 0$. □

Corollary 3 (The cap for a given bound). *Setting $B_{max} = k(G - (3 - 2/k)L)$ makes Algorithm 1 offer the service guarantee G, whatever the budget shape below the cap and whatever the base policy A. The choice is available when $G \geq (3 - 2/k)L$, and gives a positive budget only when the inequality is strict. At equality $B_{max} = 0$ and the guard is FCFS.*

Proof of Corollary 3. Substitute $B_{\max}$ into (12). □

The per-job guarantee also bounds the largest flow time. On k non-preemptive servers, FCFS keeps the largest flow time within a factor $3 - 2/k$ of the smallest achievable value. Bender, Chakrabarti and Muthukrishnan stated this bound and its tightness without proof [100]. Mastrolilli, who calls FCFS FIFO, proved the bound and showed that it is asymptotically tight [101]. The proof of Corollary 4 repeats the bound in our notation for completeness. Write $F_i = W[i] + C_i$ for the flow time of job i and F^* for the smallest largest flow time $\max_i F_i$ that any schedule of the input on k servers achieves, with or without interruption.

Corollary 4 (Largest flow time). *Under the cap of Corollary 3, every job has $W_P[i] + C_i < (3 - 2/k)F^* + G$.*

Proof of Corollary 4. We show $W_{\mathrm{FCFS}}[i] + C_i \leq (3 - 2/k)F^*$, and adding the guarantee G gives the claim. Let S_i be i's start under FCFS. If $S_i = a_i$, then $F_i = C_i \leq F^*$ and there is nothing to show. Otherwise, let t' be the supremum of the instants $t \leq a_i$ at which a server is idle after the dispatches of t. Such instants exist, since every server is idle before the first arrival. All k servers are busy on (t', S_i), since before a_i none is idle after t', and from a_i on, i waits. A job that arrived before t' has started by t', for otherwise it would wait at an instant of that set, at t' or arbitrarily close below it, while a server is idle. The jobs of this kind that still run after t' all run at one such instant, where a server is idle, so there are at most $k - 1$ of them. That busy time is filled by these jobs, each with remaining cost at most $p = \max_j C_j \leq F^*$, and by jobs that arrived in $[t', a_i]$ and started before i, of total cost X. Hence, $k(S_i - t') \leq X + (k - 1)p$. Every schedule must run these jobs and i after t' and finish them by $a_i + F^*$, so $X + C_i \leq k(a_i + F^* - t')$. Together, $S_i - a_i \leq F^* - C_i/k + (1 - 1/k)F^*$, and $F_i \leq (2 - 1/k)F^* + (1 - 1/k)C_i \leq (3 - 2/k)F^*$. □

At $k = 1$ the factor is 1, so FCFS minimises the largest flow time on one server and the guard stays within G of that optimum.

The proof uses the budget only through the cap, so the shape below the cap is free. The age-relative shape with one constant B_0 for every job also gives a multiplicative bound, $(1 - \eta)\, W < W_{\mathrm{FCFS}}[i] + B_0/k + (3 - 2/k)L$. The queue-length shape does not, by a margin that grows with γ (Supplementary Sections S4.5 and S4.6). Neither constant of Theorem 3 can be improved. With a constant budget, the term $B_{\max}/k$ is reached exactly. The term $(3 - 2/k)L$ is the exact supremum at every k over instances and budgets, and a cascade with a constant budget larger than L approaches it (Supplementary Section S8.3). Suppose that the service enforces a run-time cap that is known at arrival. The guard can then charge the cap at the start and refund the difference at completion, and this removes L from the constant (Supplementary Section S7.6). The limit L is such a cap. When L is charged this way on $k \geq 2$ servers, the guard keeps the same G. At every decision, whatever the history and the proposed job, it then overrides the base policy only if Algorithm 1 would, and at some decisions it does not. Here, to override means to start a job other than the one that the base policy proposes. Finally, a guard that never fires leaves

the base policy's run unchanged. An operator can check this on that run before deploying the guard (Supplementary Section S4.2).

6.6. A Maximum Waiting Time Gives the Same Guarantee

Recent inference servers guard against starvation by promoting a request once its wait passes a threshold [76], or once its program has waited longer than a multiple of the program's service so far [77]. Neither paper states a bound. In the simplest form, the earliest-arrived waiting job is started next once it has waited θ. Until then, the base policy chooses. We call this rule Timeout(θ). It uses arrival times and a clock, never a cost.

Proposition 1 (Timeout bound). *Fix a base policy and $\theta \geq 0$. For every input and every job i with at least one overtaker, Timeout(θ) gives $In_i < k(\theta + L)$, and every job waits*

$$W \;<\; W_{\mathrm{FCFS}}[i] + \theta + (3 - 2/k)L. \tag{13}$$

Proof of Proposition 1. Consider a start decision at a time $t \geq a_i + \theta$ while i still waits. The earliest-arrived waiting job arrived no later than i, so it has waited at least $t - a_i \geq \theta$, and the rule starts it. That job has rank at most i, so no overtaker of i is started from $a_i + \theta$ on, and every overtaker starts in $[a_i, a_i + \theta)$. Of the overtakers that one server starts in this interval, each except the last finishes before the next starts, so their costs add up to less than θ, and the last adds at most L. Summing over the servers, $\mathrm{In}_i < k(\theta + L)$. Then Theorem 1 with $\mathrm{Out}_i \geq 0$ gives $k\,(W - W_{\mathrm{FCFS}}[i]) < k(\theta + L) + 2(k-1)L$, and dividing by k gives (13). A job without overtakers has an excess of at most $(2 - 2/k)L$. Supplementary Section S4.11 relates the rule to the guard. □

At $\theta = G - (3 - 2/k)L$, the clock gives the bound that the guard gives at $B_{\max} = k\theta$. Unlike the constant of Theorem 3, the constant of Proposition 1 is not shown to be tight. The worst ratio of excess to bound that our search found is 0.996 at $k = 1$, and 0.756 to 0.829 for $k = 2$ to 4 (Supplementary Section S4.11). Section 8.4 therefore also compares the two rules at equal realised worst excess. The rules differ in what they charge a waiting job for. The clock charges it for all the work that the servers do while it waits. The guard charges it only for the completed work of jobs that arrived after it. Near saturation, a job that arrives behind a backlog is charged for the backlog. The clock then fires for most of the queue, and the schedule falls back to arrival order. This happens at the loads where ordering by cost helps most.

The two rules can be combined. The combined rule starts the earliest-arrived job if it has waited θ, and otherwise runs Algorithm 1. Both bounds then hold at once. The order matters, since serving the guard's eligible job first can pass over a job that has already waited θ (Supplementary Section S4.11). The clock part of the rule uses no completion report, so the combined rule keeps (13) when reports arrive late or never.

6.7. An Absolute Bound from the Arrival Log

A bound relative to FCFS is a margin over a schedule that is not run. The log of arrivals and their executed costs gives an absolute bound, which an operator can state as a number of seconds. Feed the same jobs to a single server of rate k, and let V_i^- be the work that this server would still hold at a_i from the jobs that arrived before i. Thus, $V_1^- = 0$ and $V_i^- = \max(0,\, V_{i-1}^- + C_{i-1} - k(a_i - a_{i-1}))$. The largest value $\sigma = \max_i (V_i^- + C_i)$ is the depth of the smallest token bucket of rate k to which the arrivals, counted in work, conform. This depth is the burst parameter of network calculus [102,103].

Proposition 2 (Absolute bound). *For every non-preemptive work-conserving policy and every job* i, $k\,W[i] \le V_i^- + (k-1)L + In_i$. *Hence,* $W_{FCFS}[i] \le (V_i^- + (k-1)L)/k$, *and under Algorithm 1 with cap* B_{max},

$$W \;<\; \frac{V_i^- + B_{\max}}{k} + \left(2 - \frac{1}{k}\right)L \;\le\; \frac{\sigma + B_{\max}}{k} + \left(2 - \frac{1}{k}\right)L. \tag{14}$$

Proof sketch of Proposition 2. Equation (10), with $\mathrm{Out}_i \ge 0$ and $\beta_i \ge 0$, gives $k\,W[i] \le R_i + \mathrm{In}_i$, where R_i is the work that the jobs of lower rank still hold at a_i. A work-conserving k-server queue never holds more than $(k-1)L$ above the single server of rate k, because whenever it holds more, all k servers are busy and it drains at the same rate k. Hence, $R_i \le V_i^- + (k-1)L$, and (11) bounds In_i under the guard (Supplementary Section S4.11). □

At $k = 1$ the fast server is the queue itself, $V_i^- = W_{\mathrm{FCFS}}[i]$, and the proposition is Theorem 3 again. Suppose that an operator wants to guarantee that no job waits longer than D, over a period whose arrivals fit a bucket of depth σ. The operator then sets $B_{\max} = kD - \sigma - (2k-1)L$: the capacity over the deadline, less the burst and less $2k-1$ time limits. The cap is non-negative only when $D \ge \sigma/k + (2 - 1/k)L$, and a shorter deadline cannot be guaranteed this way. The clock obeys the same bound with $k\theta$ in place of $B_{\max}$. Proposition 2 is proved in Supplementary Section S4.11. Sample-path comparisons of a multiserver queue with a single fast server need care. Wolff showed that some previously derived sample-path upper bounds on the work in a multichannel queue are false [104]. Our comparison keeps the additive margin $(k-1)L$, which the proof above needs.

6.8. Completion Reports That Arrive Late

Theorem 3 assumes that a completion is charged before the next dispatch decision. In a distributed service the servers are separate machines that report their completions to the scheduler over a network, and a server can take its next job before its report has been counted. Suppose that every report reaches the scheduler's counter within δ of the completion, while servers still take a job at the instant when they become free.

Proposition 3 (Late charges). *Under Algorithm 1 with a shared counter and charges delayed by at most* δ, *every job* i *with an overtaker has* $In_i < B_{max} + kL + k(L + \delta)$, *and*

$$W \;<\; W_{\mathrm{FCFS}}[i] + B_{\max}/k + (4 - 2/k)L + \delta. \tag{15}$$

For $\delta > 0$ *the bound is attained in the limit at* $k = 1$. *At* $k = 2$, *constructed instances with integer sizes, budgets and delays fall short of it by two units at every size, so the bound is attained in the limit there, also. If each server applies its own completion report to the shared counter in the same step as its next pull, and only other servers' reports travel late, Theorem 3 holds unchanged for every* δ, *lost reports included*.

Proof sketch of Proposition 3. When the last overtaker starts, the charged part is below the budget, as before. The uncharged overtakers are those still running, as in Theorem 3, and those that finished within the last δ and whose report has not arrived. On one server the latter are at most one job that started before that window and jobs that fit inside it, so they hold less than $L + \delta$. When a server reports before it takes its next job, an uncounted completion can belong only to a server that finished at the decision instant itself and has not yet taken a job, so it holds nothing in service. Supplementary Section S4.12 gives both proofs and the instances that attain the bound. □

At $k = 1$ any $\delta > 0$ already costs a whole L, because the single server takes its next job at the instant when it finishes one, before that job's report can arrive. The remedy is a change of protocol, not a smaller budget. If every server counts its own completion before it takes the next job, Theorem 3 holds again. The combined rule of Section 6.6 also keeps its clock bound for any delay. The model counts a dispatch as instantaneous. A server that must wait a network round trip for its next job is idle while jobs wait. The bound of a job then grows by two terms. One is the job's own delay between its dispatch and its start. The other is the server-idle time accumulated while some arrived job had not started, counted from the last instant at which the pool held no unfinished work, divided by k. Idle time counted only while the job itself waits does not suffice, because idling before the job arrives leaves a backlog that it inherits (Supplementary Section S4.10). Suppose instead that each server keeps its own view of the counter, in which it sees its own completions at once and those of other servers within δ. The excess is then below $B_{\max}/k + (3 - 2/k)L + (1 - 1/k)(L + \delta)$, and at $k = 2$ this bound is also attained in the limit (Supplementary Section S4.12).

The guard's counters need no coordination beyond the reports. At any instant, over$[q]$ is the total cost of the reported completions of jobs that arrived after q. It is a function of the set of reports received and of the arrival order, and not of the order in which the reports came. Suppose that two replicas of the scheduler share the arrival order and the dispatch history, and have received the same reports. They then hold the same counters and the same set of eligible jobs. They agree whenever the guard fires, and on every decision when the base policy is deterministic given the same history. The bound needs only the first agreement. A replica that lags the reports by at most δ is a scheduler whose charges arrive up to δ late, the case of Proposition 3. A report counted twice can only raise over$[q]$. The proof of Theorem 3 uses only one fact: the counted total of i was below its budget when i's last overtaker started. The true total was then also below the budget, so the bound holds with duplicated reports.

6.9. Which Assumptions Can Be Dropped

The global limit L can be replaced by the largest cost among the jobs that have arrived so far, read at the right instant. This is a bound after the fact, since a job's cost is revealed only at its completion. Servers can run at different speeds, with the budget divided by the total capacity. A fixed setup paid before every job is absorbed into its cost. Servers can also fail and destroy the progress of the jobs that they run. The bound then holds in capacity-time, with L added for each failure in the stretch of congestion around the job (Supplementary Section S7.8). Work conservation cannot be dropped. If a server can idle while jobs wait, the identity fails by exactly the forced idle time. Supplementary Section S4.10 states these relaxations with their sharp forms.

6.10. A Floor of Order L, and What a Guarantee Costs

No positive budget removes the last L of the additive constant, because a job can be displaced before anything has completed, and at that instant a guard that charges completed work has nothing to charge.

Proposition 4 (Floor). *Fix a budget rule that assigns a positive budget to every waiting job at the first dispatch decision. Then there are a base policy A that observes a job's cost only at its completion, and an input, such that Algorithm 1 run around A with that rule leaves some job with $\mathrm{excess} \geq L$. With the remaining cases, treated in Supplementary Section S4.8, every budget rule of the family (7) falls into one of two classes. Either it makes the wrapper FCFS around every base policy, or it cannot guarantee a bound $G < L$, neither around every base policy nor around one prediction-ranked base policy under arbitrary predictions. Theorem 3 is stated in this generality.*

https://doi.org/10.3390/math14214020

Proof sketch of Proposition 4. Put $k+1$ jobs of cost L into an empty system at $t=0$. Nothing has completed, so no job is eligible, and the guard accepts the base policy's choices. A base policy that starts the last-arrived job in the first phase then delays, by L, a job that FCFS starts at once. Supplementary Section S4.8 gives the proof, shows that the hypothesis cannot be dropped, and treats the rules of the family that it does not cover. Those whose budget is identically zero are FCFS, and every other one admits an input with excess at least L. □

On one server the bound is exact. With a constant budget $B>0$, the bound $B+L$ of Theorem 3 is the exact supremum of the excess (Supplementary Section S8.4). The guarantee also caps how much reordering can gain at all.

Theorem 4 (Price of the guarantee). *Take m jobs of cost L and n jobs of cost $c<L$, the long ones arriving first, all at $t=0$, and let P be a non-preemptive work-conserving policy with $\mathrm{excess}_P[j]\le G$ for every job. Write $\overline{W}_Q$ for the mean wait under a policy Q, and define $D_0=mn(L-c)/k$, $H=2(1-1/k)L(m+n)$ and $\alpha=(kG+(3k-2)L)/(nc)$. If $D_0>H$, equivalently $mn(L-c)>2(k-1)L(m+n)$, then*

$$\frac{\overline{W}_{\mathrm{FCFS}}-\overline{W}_P}{\overline{W}_{\mathrm{FCFS}}-\overline{W}_{\mathrm{SJF}}}\le \alpha+\varepsilon_+,\qquad \varepsilon_+=\frac{H(1+\alpha)}{D_0-H}. \tag{16}$$

For fixed k, L, c and G, the finite-sample term is $O((m+n)/(mn))$.

Proof sketch of Theorem 4. Call a pair of jobs inverted when the later-arrived one starts first. Summing Theorem 1 over all jobs shows that the total saving of P over FCFS equals, up to H, the sum over inverted pairs of the earlier job's cost minus the later job's cost, divided by k. This is the interchange argument behind Smith's rule [87]: on one server, for jobs that are all available at the start, ordering by increasing cost minimises the mean wait. Only a short job passing a long one contributes positively, by $L-c$. For each long job, Corollary 2 caps the short work that passes it at $kG+(3k-2)L$, which bounds the numerator. SJF inverts all mn long–short pairs, which gives the denominator. Supplementary Section S8.5 gives the proof. □

The leading term α is the capacity times G, plus $(3k-2)L$, divided by the total cost nc of the short jobs, which SJF moves ahead of the long ones. It is linear in G and reaches 1 when $kG+(3k-2)L$ covers nc. For $k\ge 2$ the finite-sample term ε_+ is positive, so the whole bound exceeds 1 already there. The bound applies to every policy with the guarantee. On this family of instances, it therefore describes the trade-off that an operator faces rather than a weakness of this guard. At $k=1$, the achievable fraction is known exactly: $\lfloor G/c\rfloor/n$, capped at 1. Whenever $kG\ge(3k-2)L$ and $D_0>H$, the guard around exact SJF meets the bound of Theorem 4 on this family, up to $2(3k-2)L/(nc)$ and finite-sample terms (Supplementary Sections S8.5 and S8.6, with an exhaustive check over 1161 parameter settings).

7. Workloads and Experimental Setup

7.1. The Workloads

We use six logs of real jobs and one log of student activity (Table 5). Our main log comes from CodeBench, the programming-course platform of the Universidade Federal do Amazonas in Brazil. The platform records every run of a student's code, and from the term 2017-2 on, it also records the run's duration [21]. A second grading log comes from ACcoding, an online judge used in university programming courses [22]. ACcoding's public release has no timestamps, only submission identifiers. We therefore keep the real

order and costs of its jobs, and draw their arrival times. In the replay, each job is charged its recorded running time plus an assumed overhead of 0.5 s. The results on ACcoding therefore test the ordering mechanism, but not waiting in seconds.

Four logs come from outside education. The Azure Functions trace of 2021 covers two weeks of calls to functions that run on demand [5,6]. Mozilla runs the tests of the Firefox browser on pools of machines. We call two of these pools CI pools A and B, and we use the waits recorded on them to check our simulator [23,24]. In Intel's Netbatch compute farm [25,26], single jobs run for days. With one limit of 5.7 days for all jobs, this farm is the case where our method does not apply. Finally, the Open University Learning Analytics Dataset (OULAD) records the daily activity of students in online courses [105]. It contains no jobs, so we use it only to check that activity also rises before deadlines outside our own grading logs.

Table 5. The logs, what each records, and what we use it for. "Arrivals" says whether arrival times are recorded, reconstructed from the recorded end time and duration, or drawn. "Waits" says whether the queueing wait of each job is recorded. Supplementary Section S2.1 compares the CodeBench counts with the statistics published with the release.

Log	Use	Size	Arrivals	Waits
CodeBench [21]	main scheduling log	4,611,325 runs in the 18 terms; 863,149 graded submissions in the development period	reconstructed	no
ACcoding [22]	ordering on a second grading platform	4,046,652 submissions, 3,237,322 outside the sealed block	drawn	no
Azure Functions 2021 [5]	a cloud workload	1,980,951 calls over 14 d	reconstructed	no
CI pool A [23,24]	simulator check; light tail	191,970 runs over 21 d	recorded	yes
CI pool B	second simulator check	30,702 runs over 7.0 d	recorded	yes
Intel Netbatch [25,26]	a case where the method does not apply	9,054,066 jobs over 30 d, 8,441,207 serial with positive run time	recorded	yes
OULAD [105]	deadline crowding elsewhere	9 course presentations, daily counts	—	—

Supplementary Section S2 audits the main log in full. Three features of the log shape how we use it. First, the timestamp of a run is the moment when its result was written. We therefore reconstruct the arrival as $a_i = t_i - C_i$, the time when the run started, and treat the recorded stamp as done_i. CodeBench runs each student's code in its own container (Section 7.2), so we take this start as the submission time. No field of the log records the delay between the two. If we read the stamp the other way, as the time of receipt, the final guard at a bound of 600 s closes more of the gap at the busiest load: 0.861 instead of 0.801 under the original protocol, and 0.839 instead of 0.705 under the online one (Supplementary Section S3.15).

Second, stamps are whole seconds, while durations are in microseconds. A model could therefore read the fractional part of the gap to the previous record and learn from it something that it should not know. Given only that part, a classifier separates heavy from light jobs with AUROC 0.8091. We therefore jitter each stamp deterministically. This AUROC then falls to 0.4499, so the fractional part no longer ranks heavy jobs above light ones. The AUROC of the forward models changes by at most 0.008.

Third, no run after 28 August 2019 lasts longer than 60 s, although the platform documents no timeout. We therefore schedule only the periods after that date. Because the recorded durations are wall-clock times on the original platform, they already contain some contention. We cap each duration at the observed ceiling, $C^{\mathrm{cap}} = \min(C, 60)$, and

impose $L = 60$ s in our pool, as a shared service would (Supplementary Table S3). For the logs from outside education, which document no time limit, we take L to be the 99.9th percentile of running times in the log's training period and cap longer jobs at it. In a deployment, the service would enforce this limit by stopping those jobs (Supplementary Table S3).

7.2. A Shared Pool Built from the Log

Our experiments run on a queue that we construct. CodeBench runs each student's code in a separate container [106], so its log records how long each run took, but not any wait in a shared queue [21]. We use the log as a record of demand, with reconstructed arrival times and recorded running times, and feed it to a pool of k servers. The pool, the queue and the scheduler are ours, and no result here describes waiting on the original platform.

Three facts support this construction. A fixed pool of judging machines or grader processes on one queue is a standard deployment [7,8]. Combining separate service systems whose service times have the same distribution makes them more efficient [107]. On this demand, the effect is large. With one container per student, 1636 containers are alive at the p99, and each of them runs code for 0.12% of its lifetime. On the same demand, a pool of 14 servers holds the p99 of waits before deadlines under 1 s (Supplementary Section S2.3).

On two CI pools, our simulator also reproduces recorded waits. We fit it on one part of each CI log. It then reproduces the other part's mean wait, hourly profile and per-job order. It does not reproduce the 90th percentile: the simulated value is 0.81 to 1.21 times the recorded one (Supplementary Section S2.3). On the grid log of the Laboratoire de Physique Corpusculaire in the Enabling Grids for E-science in Europe project (LPC-EGEE) [108], the fit is weak: it underestimates mean waits by 24% and 39% (Supplementary Section S6.4).

Arrivals are replayed at their recorded times, so the replay is open-loop. A scheduler is therefore neither rewarded with work that it would attract nor penalised for users that it would drive away. All results hold for a fixed amount of offered work. Section 8.8 restores part of this feedback. It replays each student's recorded delay between one result and the next submission, alone and with a stated model of abandonment.

A single course in a single term does not load even one server, so we overlay several. The six primary development terms are 2020-ERE, 2020-2, 2021-1, 2021-2, 2022-1 and 2022-2. They contain 40 class-terms and 402,358 of the 863,149 graded submissions of the development period (Table 5). The 400,790 of them with a positive running time are the jobs. The other five development terms, 2018-1 to 2019-2 and 2020-1, serve only to fit the predictors. A class-term is the set of submissions of one class in one academic term. We superimpose 44 copies of these class-terms, aligned by weekday and hour. A fixed rule sets this count. It is the smallest number of copies at which every overlay needs at least 4 servers at full load. At that count, the three loads must also give three different server counts. The same rule, applied unchanged, gave 31 copies on the sealed terms. Each copy of each class-term is shifted by a whole number of weeks, drawn uniformly from 0 to 11. Overlay r uses the seed $300 + r$. We draw the shifts five times and obtain five such traces, which we call overlays. Overlaying whole terms keeps the daily and weekly cycles and each job's position relative to its own deadline. A common way to raise the load is to compress the gaps between arrivals [109]. However, this would also compress the daily cycle [110]. In all, 17,634,760 jobs arrive over 30 weeks, and 4,219,556 of them arrive in the 24 h before an assessment deadline.

We run each overlay at three loads, which differ only in the number of servers. For a target utilisation ρ, we take the executed work, in seconds, that arrives in the overlay's

busiest whole hour. We divide it by 3600ρ and round the result to the nearest integer, which gives k. Four of the five overlays use $k = 8, 5$ and 4. On the fifth, whose busy hours carry the least work, k is 7, 5 and 4. Tables that average over overlays therefore show $k = 7.8$ at the lightest load. Averaged over the five overlays, the utilisation of the busiest hours is 0.502, 0.783 and 0.979, and we call the three loads $\rho = 0.5$, 0.8 and 1.0.

The five overlays are one construction and not five platforms. Our intervals are 95% percentile intervals from 2000 resamples of whole weeks, drawn with seed 20260919. We draw these resamples once and use them for every policy, so that differences between policies are paired. Because they treat the 30 weeks as the independent units, these intervals cover variation over time but not the choice of terms or courses. We also resampled the 40 whole class-terms instead, 100 times, and rebuilt the overlays each time. This widens the intervals several-fold (Supplementary Tables S18 and S38). Under the online protocol (Section 8.1), the interval of Guard(600) at the busiest load then becomes [−0.043, 0.894] around 0.705. Because this interval contains zero, a different set of classes and terms could leave the guard with no gain at the busiest load. The guard's difference from unguarded ordering is -0.158, and the interval of this difference, $[-0.317, -0.053]$, still excludes zero.

Arrivals follow the course calendar. When the number of runs per class is predicted a week ahead, adding calendar features to the recent counts lowers the log-scale RMSE from 1.243 to 1.112, a change of -0.1307 $[-0.1395, -0.1213]$. OULAD shows the same pattern outside our grading logs. In each of its nine course presentations, the three days before an assessment carry 1.29 to 2.95 times the activity of other days. With calendar features, a warning of peak activity seven days ahead reaches an F1 score (the harmonic mean of precision and recall) of 0.593, against 0.000 for a moving average of the previous two weeks. We archive the forecasting models, the definition of a peak and the outputs with the code (Data Availability Statement).

7.3. Development Data, Sealed Data and Parameter Selection

Three logs have a part that was sealed before the method was designed. These parts are three academic terms of CodeBench (2023-1, 2023-2 and 2024-1), the last 20% of ACcoding's submission identifiers, and OULAD's 2014 year. Before the method was frozen, the sealed parts were read only for bookkeeping. We parsed and counted the CodeBench archives and computed their static code features. This is why the run count of Table 5 covers all 18 terms. On loading, we dropped the 809,330 sealed ACcoding rows after recording their count and first identifier. In an audit under an earlier project plan, we counted OULAD's 2014 requests. No cost, prediction or scheduling result was computed on a sealed part before the freeze, and every access is recorded in the repository.

We evaluated the method on the sealed CodeBench terms once, on 25 September 2026, after the method, its parameters and the primary metric had been frozen. Section 8.2 reports that evaluation, together with a frozen evaluation of ACcoding's sealed part on 3 October 2026. Every other number in this paper is a development number. A first attempt at the sealed run stopped before producing any result, because a cleaning rule named only the development terms. We extended the rule to the sealed terms and froze the protocol again (Supplementary Section S10).

We chose the guard's shape parameters by a fixed rule on validation cells. Each of the three shapes has a grid of parameter points, and the rule keeps the points whose worst harm over 15 validation cells is at most $G/2$. Among these points, it takes the one whose gap closed in the worst cell is largest (Supplementary Section S5). The validation cells are the five overlays at three loads, built on development terms other than the last.

Table 6 lists the data that each stage used. The validation overlays leave out the last development term, and we chose the shape parameters on them. The primary pool of five

overlays gives every development number. The sealed terms were used once, as the final test, and nothing was chosen on them.

Table 6. Data used at each stage. The overlays are built from CodeBench development terms as in Section 7.2, and the primary pool and the validation overlays share all terms but the last.

Stage	Data	Chosen or Reported
Fitting the predictors	terms from 2018-1 up to the term before each target term; the targets are the six primary development terms: 2020-ERE, the emergency remote term (Ensino Remoto Emergencial) that preceded the delayed 2020-1, and 2020-2 to 2022-2	model weights and scores, frozen before each target term starts
Exploration	primary pool: five overlays of the six primary development terms at three loads (15 cells)	the guard family and a coarse grid, before the selection rule was written
Validation	five validation overlays without the last development term (2022-2), at three loads (15 cells)	the shape parameters of Guard(G), by the rule above
Development evaluation	primary pool (15 cells)	every number of Section 8 except the sealed evaluation
Final test	sealed CodeBench terms 2023-1, 2023-2 and 2024-1, opened once on 25 September 2026	nothing chosen; Section 8.2

Because we explored the guard family and a coarse grid on the primary pool, the development numbers are not held out, and we do not call them out of sample. The selection rule is sensitive to the set of cells, and the five overlays are week-shifts of the same class-terms. On a single server, the queue-length shape selected at $G = 1200$ s on cells with $k \geq 4$ does not carry over, although the age-relative points selected on the same cells do (Section 8.6). We therefore do not call the parameters validated. Only the benefit depends on them, and the guarantee does not.

As a sensitivity analysis, we repeated the selection on the validation overlays alone, under the online replay of Section 8.1. This time the rule chose among all families at once. They were the three guard shapes, Skip, Timeout, linear aging (Section 8.4 and Supplementary Table S11) and ordering by score within blocks of a fixed number of arrivals. At each of the three bounds, the rule chose a guard shape: the age-relative shape at $G = 300$ s, the queue-length shape at 600 s, and a shape with both terms at 1200 s. On the primary pool, these choices close between 0.046 less and 0.041 more of the gap than Guard(G) at the same bound and load, and every job stays within its bound. The validation overlays share all terms but the last with the primary pool (Table 6). This repeated selection therefore checks the choice on overlapping development data, and it does not separate the choice from the exploration. The test on data that the exploration did not use is the sealed evaluation of Section 8.2. Supplementary Section S5 records what the original rule selected, which results the released code reproduces, and how the choice changes when one validation cell is dropped.

7.4. Software and the Use of Generative AI

We ran the simulations and analyses with Python 3.12.13, NumPy 2.5.3, SciPy 1.18.1, pandas 3.0.6 and Numba 0.67.0. The tree models use LightGBM 4.7.0 [18], and the neural models use PyTorch 2.11.0. For the proofs of Section 6 we used Lean 4 at version v4.35.0-rc4 with Mathlib at commit 1f41440. We release the code under the MIT licence in the archive named in the Data Availability Statement.

We used two generative AI tools in this study. They were Claude Code (Anthropic, version 2.1, with the Claude Opus 5.5 and Claude Fable 5.1 models) and Codex CLI (OpenAI, version 0.163, with the GPT-5.6-Sol and GPT-6-Astra models). We used these

tools to edit the text, to write, run and review the code, and to run experiments and analyse their results. We also used them to write the Lean proofs, to search the literature and check the references, and to search for counterexamples to the stated bounds. The Lean kernel checks every machine-checked proof. For each number printed in the paper that the released code produces, a script in the archive compares the printed value with that code's outputs. Supplementary Section S5.4 lists the results that come from archived analyses instead.

8. Experiments

Every policy is simulated on the shared pool of Section 7.2, with the same arrivals and costs, so each comparison is made job by job. Supplementary Table S4 lists the configuration. The confirmatory result is the evaluation on the sealed terms (Section 8.2), whose policies, parameters, protocols and metric were frozen before those terms were opened. Every other contrast in this section and in Section 4 is exploratory. Each 95% interval covers its own comparison and is not adjusted for the number of comparisons made.

8.1. Development Results Under the Online Protocol

Suppose that each score may use the results from its own copy of a class-term only once the simulated queue has produced them. The guard then keeps most of the benefit of ordering, and every job stays within its bound. In the prediction study of Section 4, a job's result becomes available at the time when the original platform recorded it. In the simulated queue, a job completes at a time that depends on the policy, so a score computed on the original clock can use a result that the simulated queue has not yet produced. The *online* protocol removes this problem within each copy of a class-term. It freezes the fitted model and simulates each policy once. When a job arrives, the model scores it from the earlier jobs in the same copy of its class-term, but only from those that this simulation has completed by that moment. Results of other classes and terms keep their recorded times. A score therefore follows the simulated queue for its job's own copy and the recorded clock for all others. A shared service would also delay the results of other classes and terms. The conservative protocol below uses none of them and is the pessimistic reference for that case. Supplementary Section S3.7 defines the simulation and checks it against a second implementation.

We compare the online protocol with four reference protocols. The *exact* protocol also freezes the model. It simulates a policy, and for each job it then withholds every result from the job's own copy that was not yet complete when the job arrived. It then rescores the jobs and simulates again, and it repeats these steps until no score uses an unavailable result from its own copy. Its name refers to this fixed point, and its running times are predicted like all others. Other classes and terms keep their recorded times, as under the online protocol. A result that the exact protocol has withheld once stays withheld, even when a later simulation completes it in time. The exact protocol can therefore use less history than the online protocol. The *original* protocol keeps the platform's clock and is optimistic. The *conservative* protocol admits a result only an hour after it appeared, uses only history from the same class-term copy, and refits the models. The *static* protocol uses no outcome history at all. The guard's parameters are the same under all five.

At the busiest load, Guard(600) under the online protocol closes 0.705 of the gap (Table 7). Unguarded SPJF-E closes 0.864 under the same protocol, but it delays one job by 5812.7 s beyond its FCFS wait (Supplementary Table S31). Resampling whole classes and terms instead of weeks widens the interval of the guard until it contains zero (Section 7.2). The online figure lies between the exact and the original ones, because the exact protocol uses less history than the online one and the original protocol uses results that the queue

has not produced. At the two lighter loads the online and original figures are closer, and their intervals overlap. Under the conservative and static protocols, the guard closes the least. The conservative protocol also withholds history that the queue would have had, and it refits the models. Supplementary Section S3.7 separates these two effects.

The parameters were selected under the original protocol. Selecting them under the online protocol instead changes the choice. We used the same rule on the same validation cells, but only among the four best feasible points of each shape under the original protocol (36 candidates). At $G = 600$ s the rule then picks the queue-length shape of Guard-queue(600) in Supplementary Section S11. On the evaluation overlays at the busiest load, it closes 0.692 of the gap under the online protocol (Supplementary Table S31). The guard as selected on the original clock closes about as much of the gap online, at more than twice the harm of Guard-queue(600). At $G = 300$ s, the rule chooses another age-relative point. At $G = 1200$ s, it chooses an age-relative point instead of the queue-length shape. We replayed both new points on the evaluation overlays. At the busiest load, the new point at 300 s closes 0.373 of the gap online, and the selected guard closes 0.367, at about the same harm. The new point at 1200 s closes 0.796 against 0.754, at about four times the harm. The full grids were run under the original protocol. The rest of this section therefore reports results under that protocol unless stated otherwise, and these figures are optimistic references. Supplementary Section S11 replays 24 of the compared policies under the online protocol. They are the two unguarded rankings, the aging baseline of Section 8.4 and, at each bound, the guard, its three single-shape winners, and Fixed, Skip and Timeout. There, every one of them closes less of the gap at the two higher loads, and no policy with a bound exceeds that bound. Every paired difference whose interval excludes zero on the original clock keeps its sign online. Four of these intervals no longer exclude zero, and three others now do. The one conclusion of this section that changes is the choice between the two rankings in Section 8.3. We also replayed four other analyses online (Supplementary Section S11): the inverted-tail score, the single server, the timestamp read as the time of receipt, and the two treatments of runs by one user that end in the same second. The other comparisons of this section use the original protocol, and so does the queue comparison on which we chose the predictor in Section 4.3.

Table 7. Guard(600) under five protocols for what the scores can use. Online: each score uses the results from its own class-term copy that the simulated queue has completed when its job arrives. It also uses the recorded results of other classes and terms. This is the protocol of the headline results. Exact: we withhold results and simulate the queue again until no score uses an unavailable result. A withheld result stays withheld. Original: the platform's own clock (optimistic). Conservative and static withhold more than the queue requires. Gap closed with 95% intervals from resampled weeks (Section 7.2), which do not cover the choice of classes and terms (Supplementary Table S38 does). Waits, largest excess and harm are in seconds. Fired: firing rate (4), the share of dispatch decisions, weighted by queue length, at which the guard fired. At $\rho = 0.5,\ k = 7.8$ is the mean over the five overlays.

Load	Protocol	$p99_{dl}$	Gap Closed	Max Excess	Harm	Fired (%)
$\rho = 0.5,\ k = 7.8$						
	online	15.22	0.725 [0.696, 0.746]	499.5	248.3	0.38
	exact	15.25	0.724 [0.697, 0.747]	498.6	248.1	0.45
	original	14.97	0.740 [0.713, 0.760]	492.2	237.0	0.17
	conservative	16.68	0.640 [0.604, 0.671]	499.9	235.2	0.94
	static	17.95	0.560 [0.519, 0.594]	498.0	225.1	0.38
$\rho = 0.8,\ k = 5$						
	online	50.57	0.800 [0.772, 0.819]	509.3	215.7	2.18
	exact	51.90	0.785 [0.748, 0.807]	509.0	215.6	2.64

	original	47.47	0.837 [0.815, 0.851]	507.8	319.6	1.31
	conservative	64.77	0.634 [0.560, 0.688]	509.4	327.7	3.26
	static	70.18	0.571 [0.504, 0.638]	512.9	341.6	3.89
$\rho = 1,\ k = 4$						
	online	111.25	0.705 [0.648, 0.768]	517.7	333.2	7.27
	exact	118.94	0.672 [0.602, 0.743]	519.9	332.8	7.82
	original	89.24	0.801 [0.764, 0.842]	523.7	339.9	7.15
	conservative	174.71	0.429 [0.357, 0.533]	527.8	303.5	7.71
	static	193.11	0.349 [0.281, 0.457]	528.1	316.1	11.18

8.2. The Sealed Terms

On the three sealed terms of the main log, the frozen guard keeps every job within its bound. Its point estimate of the gap closed is larger than on the development terms, although the class-term intervals of the two pools overlap (Section 7.2). The terms were evaluated once, on 25 September 2026, after the method, its parameters and the primary metric had been frozen (Section 7.3). An earlier attempt stopped before writing any result: a cleaning rule named only the development terms, so it did not remove a zero service time. The rule was extended before the protocol was frozen again. The run replayed the frozen configuration and code, and it selected nothing. The access ledger in the repository lists each of its ten commands and what each produced. The sealed terms hold fewer class-terms than the development pool, so each of their five overlays superposes 31 copies over 29 weeks. The number of servers follows the rule of Section 7.2, and the busiest hours reach utilisations of 0.487, 0.770 and 0.954 at the three loads.

At the busiest load, Guard(600) under the online protocol lowers the $p99_{dl}$ from 187.20 s under FCFS to 47.22 s and closes 0.894 [0.862, 0.912] of the gap (Table 8). This interval comes from resampled weeks, as do the intervals of the main tables. Resampling whole class-terms of the sealed terms instead, after their result was known, gives [0.116, 0.933], which excludes zero (Supplementary Table S39). On the development terms, the same resampling widens the interval of this guard to include zero (Section 7.2). The guard closes 0.880 under the exact protocol and 0.911 under the original one, and the five protocols fall in the same order as on the development terms. Unguarded SPJF-E closes 0.936 [0.909, 0.948] online. The guard's largest excess over FCFS stays below its bound of 600 s, whereas SPJF-E delays one job by 3301.6 s and has a harm of 1602.0 s. No guard in the main comparison exceeds its bound at any load (Supplementary Section S10).

On the cells with several servers, every policy closes a larger share of the gap on the sealed terms than on the development ones, except five of the deliberately corrupted rankings, which do more damage there. The sealed run therefore confirms the guarantee and the order of FCFS, the guards and SPJF-E. At every load, each guard closes more of the gap than FCFS and less than SPJF-E, as on the development terms. Guard(600) and Guard(1200) change places at the two lighter loads (Supplementary Table S28). The sealed run does not confirm the size of the development figures. The running-time predictor ranks the jobs of the sealed terms as well as those of the development terms. The pooled AUROC of SPJF-E's score is 0.9414 [0.9349, 0.9470] on the sealed terms and 0.9360 [0.9289, 0.9424] on the development terms. The two intervals overlap.

The larger point estimate on the sealed terms is consistent with the primary metric and not with the load. At the busiest load, the guard closes 0.189 more of the gap on the sealed terms under the online protocol. Unguarded SPJF-E closes 0.072 more. The guard's cost against SPJF-E falls from 0.158 to 0.041, and this gives the other 0.117. One cell from each pool has four servers, nearly the same busiest-hour utilisation and nearly the same gap between FCFS and SJF. The guard still closes 0.681 of the gap in the first and 0.889 in

the second. The primary metric counts only the jobs that arrive in the 24 h before a deadline. These jobs are 37.5% of the sealed jobs, against 23.9% of the development ones. On the 99th-percentile wait over all jobs, the same ratio is 0.904 for the guard on the development terms and 0.896 on the sealed terms. On the mean wait, SPJF-E closes less on the sealed terms at all three loads (0.819 against 0.903 at the busiest).

Table 8. Guard(600) under the five protocols of Table 7 on the three sealed terms, evaluated once after the method and its parameters were frozen. Columns are as in Table 7, and k is the mean over the five overlays.

Load	Protocol	p99$_{dl}$	Gap Closed	Max Exc.	Harm	Fired (%)
$\rho = 0.5,\ k = 8.2$						
	online	3.76	0.895 [0.872, 0.906]	260.8	166.2	2.59
	exact	3.77	0.893 [0.871, 0.903]	256.6	166.6	3.29
	original	3.75	0.897 [0.878, 0.907]	264.0	173.9	4.01
	conservative	4.15	0.839 [0.816, 0.858]	264.1	178.0	3.40
	static	4.46	0.792 [0.744, 0.824]	259.3	203.3	0.16
$\rho = 0.8,\ k = 5.2$						
	online	24.59	0.884 [0.862, 0.903]	506.6	202.6	6.40
	exact	24.74	0.880 [0.860, 0.899]	499.2	202.6	4.41
	original	24.22	0.891 [0.874, 0.908]	504.6	189.5	7.01
	conservative	28.09	0.811 [0.772, 0.845]	520.3	233.2	7.06
	static	31.70	0.740 [0.691, 0.786]	533.7	255.3	1.33
$\rho = 1,\ k = 4.2$						
	online	47.22	0.894 [0.862, 0.912]	533.8	237.5	7.65
	exact	49.40	0.880 [0.839, 0.903]	526.5	243.1	5.25
	original	44.33	0.911 [0.890, 0.926]	522.9	293.2	7.10
	conservative	65.08	0.784 [0.712, 0.845]	518.8	244.6	8.57
	static	82.66	0.679 [0.583, 0.783]	528.6	332.3	3.20

The sealed terms come from the same platform as the development data. They test the frozen method on new classes and terms of one service, not on a second service. A second platform, ACcoding, also has a sealed part, its last 20% of submission identifiers (Section 7.3). On that part the guarantee held for every job, and the gain did not carry over. We evaluated this part with a frozen protocol on 3 October 2026, in a run that read those rows for the second time. The first attempt, on 2 October, stopped at its first trace before producing any result. We then renewed the protocol lock, and the only change was to the online-scoring code. Guard(600), scaled to ACcoding's limit of 30.5 s, has the bound 305 s, and its largest excess was 257.2 s. In 145 million checks, one per job in each guarded run, no job exceeded the bound. Here, the score is SPJF-log, a log-scale model fitted on ACcoding, and the guard wraps it. Unguarded SPJF-log made the p99 worse than FCFS in five of the six cells, with a gap closed as low as −2.30. By contrast, the gap closed by Guard(600) stayed between −0.17 and 0.01. This run cannot separate the platform from the load. ACcoding's released data carry no submission times, so the protocol places submissions around contest start times and rescales the gaps to reach a target load. The contest list ends in November 2021, so at least 23% of the sealed submissions fall at one instant, and the rescaling stopped at its limit. The six cells therefore ran at 1.8 to 2.2 times their target load. The run thus measures the guarantee on a second platform, but not the gain at a calibrated load. Supplementary Section S13.2 gives every cell.

8.3. What to Rank by

The two scores give almost the same benefit, and which of them is ahead depends on the protocol. On the original clock, ranking by expected cost beats the log-scale score at the two higher loads, by $+0.012$ [0.004,0.019] and $+0.019$ [0.011,0.024] of the gap. At the lightest load, the two scores tie, at -0.000 [-0.016,0.013]. Refitting the log-scale model from scratch changes this difference by at most 0.001 (Supplementary Table S16). Under the online protocol the log-scale score is ahead at the lightest load, and the two tie at the higher loads (Supplementary Table S32). Both scores are replayed with their own frozen models, so the change comes from the history that each can use. The choice of SPJF-E as the base ranking therefore rests on the original protocol. On the original clock, quantile scores and squared error on raw seconds lose to the Tweedie fit at every load. A rule that only separates predicted-heavy from predicted-light jobs recovers little of the benefit, so most of the benefit comes from the order inside the light class (Supplementary Section S3.10 and Table S14).

The two kinds of prediction error do not cost the same. We repaired the scheduler's own score, SPJF-E, on the five overlays at the three loads, under the original protocol (Supplementary Section S3.16 gives the waits). Replacing the largest 1% of underestimates by the true cost closes 0.84 to 0.86 of the distance to SJF in $\mathrm{p99}_{\mathrm{dl}}$ at $\rho = 1.0$. At the lighter loads, it closes 0.58 to 0.78. Repairing the largest 1% of overestimates closes 0.18 to 0.27 of that distance. Treating a long job as short is thus the expensive mistake in all fifteen cells. The factor is 2.8 to 4.1 at the 1% repair and 3.7 to 11.7 at the 0.1% repair. The repair depends on the score's numerical scale, so this result describes our fitted predictor and not prediction errors in general. On the log-scale score, the factor is 3.5 to 3.7 at the 1% repair on the two overlays that we tested at $\rho = 1.0$. The measure of score quality that we use next takes this error into account, because it measures how well a score ranks the longest jobs above the rest.

On the main log, the benefit falls steadily as the scores get worse, and the guarantee does not depend on them (Figure 4). We degraded two scores, the true cost and the stored SPJF-E score, by multiplying each with $e^{\sigma Z}$ for a standard normal Z and σ from 0 to 3, and we added a random score. We ran each score unguarded and under the three guards, on the 15 cells of Table 9 with the noise seeds 20261018 to 20261020. The random scores use the seeds 3, 1003 and 2003. The horizontal axis is the AUROC with which a score ranks the longest 1% of jobs above the rest. At the heaviest load, Guard(600) closes 0.970 of the gap with the true costs and 0.054 with random scores. The guard also keeps a smaller share of the unguarded benefit as the score deteriorates. It keeps 0.87 of it with the SPJF-E score and 0.16 with random scores. Over the three seeds, the second figure ranges from 0.14 to 0.17. For this noise family and these three seeds, a better score thus helps the guarded policy twice. It raises both the unguarded benefit and the share of it that the guard keeps. In every run, the worst job used at most 0.944 of its bound, and the per-job checks found no violation.

Table 9. Guard(G) at three bounds and three loads, original protocol, with the FCFS and SPJF-E figures of each load in its header. Gap closed with 95% intervals. Reduction is the plain percentage reduction of $p99_{dl}$ against FCFS, which differs from the gap closed. Max excess and harm are the worst over the five overlays, and the other columns are means. Times in seconds. Fired as in Table 7. Exact SJF, which knows every running time, has a largest excess of 1034.0, 2667.4 and 4755.5 s at the three loads. Supplementary Table S6 adds the comparators.

Load	Policy	G (s)	$p99_{dl}$	Gap Closed	Reduction (%)	Max Excess	Harm	Fired (%)
$\rho = 0.5$, $k = 7.8$; FCFS $p99_{dl}$= 27.64 s, SPJF-E 14.68 s with max excess 1241.4 s								
	Guard(300)	300	15.58	0.705 [0.678, 0.724]	43.7	192.5	132.1	1.35
	Guard(600)	600	14.97	0.740 [0.713, 0.760]	45.8	492.2	237.0	0.17
	Guard(1200)	1200	14.78	0.750 [0.723, 0.772]	46.4	996.7	106.0	0.09
$\rho = 0.8$, $k = 5$; FCFS $p99_{dl}$= 119.35 s, SPJF-E 43.33 s with max excess 3044.5 s								
	Guard(300)	300	55.94	0.738 [0.697, 0.761]	53.0	209.7	114.1	11.57
	Guard(600)	600	47.47	0.837 [0.815, 0.851]	60.1	507.8	319.6	1.31
	Guard(1200)	1200	46.31	0.850 [0.829, 0.865]	61.1	1096.1	117.7	0.41
$\rho = 1.0$, $k = 4$; FCFS $p99_{dl}$= 273.50 s, SPJF-E 62.79 s with max excess 5904.5 s								
	Guard(300)	300	163.40	0.478 [0.387, 0.602]	40.2	229.1	134.8	28.64
	Guard(600)	600	89.24	0.801 [0.764, 0.842]	67.3	523.7	339.9	7.15
	Guard(1200)	1200	79.32	0.844 [0.817, 0.870]	70.9	1110.4	123.4	1.40

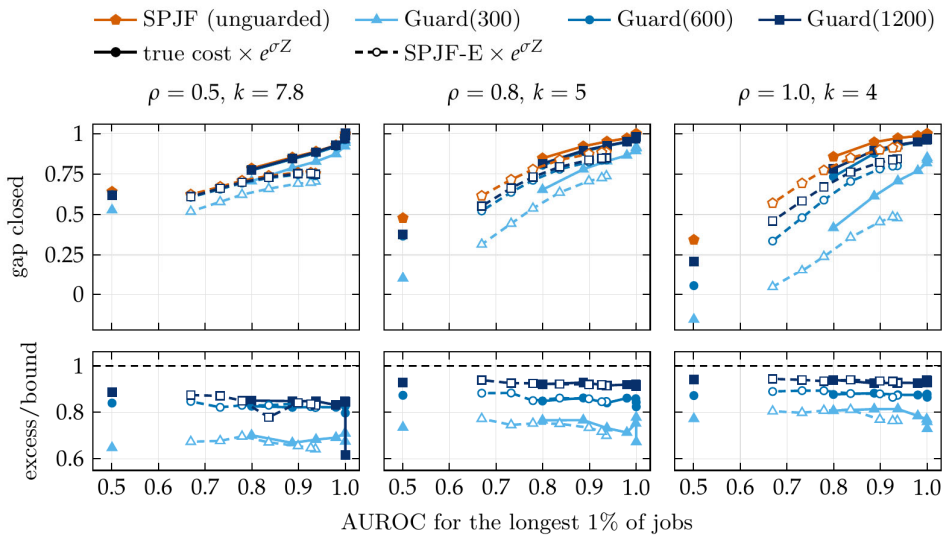


Figure 4. Benefit against prediction quality, with the bound held. The horizontal axis is the AUROC with which a score ranks the longest 1% of jobs above the rest. Upper row: gap closed by the unguarded ranking (SPJF) and by Guard(G) when the score is the true cost (filled marks, solid lines) or the stored SPJF-E score (open marks, dashed lines), multiplied by $e^{\sigma Z}$ with standard normal Z and $\sigma = 0$, 0.25, 0.5, 1, 1.5, 2 and 3; $\sigma = 0$ gives SJF's order and SPJF-E, and the isolated marks near 0.5 are random scores. On true costs, $\sigma \leq 0.5$ keeps the AUROC at or above 0.999, so those three settings share the right edge. Lower row: for the guarded runs, the largest ratio of a job's excess over FCFS to the excess that its own bound allows. The largest of the 135 values is 0.944, and the per-job bound checks inside the runs find 0 violations. Gap closed is the mean over five overlays and three noise seeds, and the ratio is the worst over them. In the legend, the two black entries give the line styles of the two score families, and the colours give the policies. Original protocol. At equal AUROC the true-cost family closes more of the gap than the SPJF-E family, so this AUROC does not capture all that a score contributes to the queue.

8.4. What the Guarantee Costs and Gains

The guarantee costs part of the gap at every bound that we tested, and in return, no job waits far beyond its FCFS wait. Table 9 gives the guard at three bounds and three loads. At $\rho = 1.0$, Guard(600) closes 0.801 of the gap, and the unguarded policy closes 0.916. A bound of ten time limits therefore costs 0.115 of the gap. The cost is 0.072 at $G = 1200$ s and 0.438 at $G = 300$ s, and none of the three paired intervals contains zero. The cost falls as the slack above the floor of Corollary 3, $G - (3 - 2/k)L$, grows relative to the $\mathrm{p99}_{\mathrm{dl}}$ of FCFS. This slack is about half of that wait at $G = 300$ s and almost four times it at $G = 1200$ s. Under the online protocol the cost is larger (Supplementary Table S32).

In return, the guarantee protects the worst job. Under the original protocol of Table 9, the unguarded policy delays one job at the busiest load by 5904.5 s beyond its FCFS wait. Exact SJF, which knows every running time, delays one job by about four fifths of that (Table 9), so most of this delay comes from ordering by length and not from wrong predictions. The guard keeps every job within its bound. It also keeps the worst wait among heavy jobs within about 450 s of its FCFS value, whereas the unguarded policy multiplies that wait by 2.8 to 4.1. This protection costs part of the reduction in the mean wait and in the waits just below the worst job. At the busiest load, the guard keeps 0.751 of SJF's reduction of the mean wait, against 0.930 unguarded. Its 99.9th-percentile wait over all jobs is 717.6 s, against 192.4 s unguarded and 984.3 s under FCFS.

The harm limit used in selection does not always carry over to the evaluation overlays. At the two higher loads, the harm of Guard(600) exceeds the 300 s validation limit (Table 9), although every excess stays within the bound. The shape of the budget matters mainly where the bound is tight and the load high. There, the shape mostly determines which jobs are delayed, and less how much of the benefit survives (Supplementary Sections S3.3 and S3.11). Fixed(G) spends the whole cap and needs no selection. On these cells it closes about as much of the gap as Guard(G). At the busiest load with $G = 600$ s, it closes 0.813 against 0.801. Its harm, 465.5 s against 339.9 s there, exceeds the validation limit at every bound (Supplementary Table S6). The selection therefore buys lower harm, not a larger gap. Counting overtakes instead of work gains almost nothing at the two higher loads (Supplementary Figure S2). Overtakers here mostly run for less than a second, but a count must be sized for the worst case of L per pass. Our baseline that counts overtakes charges each pass when the overtaker starts. Charging it at completion instead changes its gap closed by at most 0.015, and by at most 0.001 at the two higher loads.

The aging baseline ranks by $\hat{C}_i - \omega(t - a_i)$, with the credit ω chosen under the same harm limit. It keeps less of the benefit than the guard and has no bound. Under the online protocol at the busiest load, it closes 0.258 of the gap, about a third of the guard's share, and one job exceeds its FCFS wait by 1808.9 s. Under the conservative protocol, the order is the same, 0.051 against 0.429 (Supplementary Tables S12 and S31). The two were not tuned alike. We chose the credit from 11 values and the guard's parameters from 258 grid points over three shapes (Supplementary Section S5.2). The credit, 0.03 score units per second of age, was selected under the conservative score, and the guard under the original one. The repeated selection of Section 7.3 put the two on one footing: one rule, the online score and the same harm limit. There, no credit met the harm limit at $G = 300$ s. At 600 and 1200 s, the best feasible credit closes 0.090 and 0.251 of the gap on the worst validation cell. The guard closes 0.591 and 0.703 there.

A maximum waiting time gives a guarantee of the same form (Proposition 1), but near saturation and at tight bounds, it keeps much less of the benefit than the guard. Table 10 compares Timeout($G - (3 - 2/k)L$) with Guard(G). This threshold gives both rules the bound G. At $\rho = 0.5$ and at $G = 1200$ s, the two differ little. At $\rho = 1.0$ and $G = 300$ s, the clock closes almost none of the gap, and its $\mathrm{p99}_{\mathrm{dl}}$ is barely below FCFS's. The clock charges a waiting job for all the work that the servers do while it waits. The guard charges

it only for the completed work of jobs that arrived after it (Section 6.6). The clock's bound is not known to be tight, so we also compare the two at equal realised worst excess. We ran Timeout with θ from 15 to 960 s, doubling at each step, on all five overlays. At the busiest load, $\theta = 480$ s comes nearest to the largest excess of Guard(600) on every overlay. Timeout then reaches 492.6 to 509.2 s and closes 0.462 to 0.509 of the gap. Guard(600) reaches 512.3 to 523.7 s and closes 0.787 to 0.816. Under the online protocol, every difference of Table 10 keeps its sign. At $\rho = 1.0$ the clock then closes 0.282 of the gap at $G = 600$ s and 0.023 at $G = 300$ s (Supplementary Table S33).

Table 10. Guard(G) against the maximum waiting time Timeout($G - (3 - 2/k)L$), which has the same bound G. Gap closed, mean over the five overlays, with 95% intervals. The difference is Guard minus Timeout, resampled as a difference. Harm is the worst over the overlays, in seconds. Original protocol.

Load	G (s)	Guard(G)	Timeout	Difference	Harm, Guard/Timeout
0.5	300	0.705 [0.678, 0.724]	0.652 [0.619, 0.679]	+0.053 [+0.030, +0.077]	132.1/148.0
	600	0.740 [0.713, 0.760]	0.734 [0.707, 0.753]	+0.006 [+0.001, +0.012]	237.0/237.0
	1200	0.750 [0.723, 0.772]	0.752 [0.724, 0.774]	−0.002 [−0.005, +0.001]	106.0/237.0
0.8	300	0.738 [0.697, 0.761]	0.244 [0.017, 0.508]	+0.494 [+0.241, +0.691]	114.1/177.1
	600	0.837 [0.815, 0.851]	0.789 [0.754, 0.807]	+0.048 [+0.030, +0.080]	319.6/443.7
	1200	0.850 [0.829, 0.865]	0.857 [0.837, 0.869]	−0.006 [−0.012, −0.001]	117.7/469.2
1.0	300	0.478 [0.387, 0.602]	0.031 [0.012, 0.055]	+0.447 [+0.369, +0.550]	134.8/182.3
	600	0.801 [0.764, 0.842]	0.420 [0.131, 0.673]	+0.381 [+0.158, +0.650]	339.9/456.7
	1200	0.844 [0.817, 0.870]	0.827 [0.786, 0.865]	+0.017 [−0.002, +0.044]	123.4/708.8

Backfilling is the wrapper that batch schedulers put around a queue order. In the two forms that we tested, it either keeps its own promise and stays at FCFS, or uses the ranking and loses the promise. Conservative backfilling (CBF) lets a job start early only if no queued job loses its reservation [56]. Slack-based backfilling lets later jobs delay a job by at most its slack beyond the start planned at its arrival [58]. The original rule computes each job's slack from its priority. It tries to place an arriving job at every instant at which the planned use of the servers changes. We give every job the same slack G and try only two start times: the current time and the earliest time at which the job fits. We try only these two because the full search takes a number of steps that is cubic in the queue length at each event. Table 11 gives both rules on the fifteen main cells, each in two forms (Supplementary Section S13.4). Each rule is held to its own promise, so the table does not compare the rules under one common guarantee. When the rules plan every job at the limit L, which no job exceeds, both keep their promises and stay at FCFS. Supplementary Section S7.6 argues this for conservative backfilling when every job takes one server. CBF started all 264.5 million jobs of the fifteen cells at their FCFS times, and slack-based backfilling started 50,183 of them at another time. The rules can instead plan on the SPJF-E predictions. A job that outlives its prediction is then re-estimated at L [70], and slack-based backfilling plans again after such an overrun, by a rule of our own. In this form, the two rules use the ranking but lose their promises. CBF then closes almost none of the gap. Slack-based backfilling closes less than Guard(600) and Guard(1200) at every load, and at $\rho = 1.0$ up to 956 jobs of one overlay wait more than 1200 s beyond FCFS. Its waits are the same for every G. The reason lies in the variant that we implemented (Supplementary Section S13.4). There, a job's slack grows whenever compression moves the job earlier, so the slack never limits the schedule.

Table 11. The guard against backfilling on the fifteen main cells, original protocol. Each entry is the gap closed, mean over the five overlays, and the largest excess over FCFS in seconds, worst over the overlays. The rules marked L plan every job at the limit L and keep the promise shown. The rules marked P plan on the SPJF-E predictions, re-estimate a job at L when it outlives its prediction and carry no bound. The slack rows are the same for $G = 300$, 600 and 1200 s.

Policy	Promise to Each Job	$\rho = 0.5$	$\rho = 0.8$	$\rho = 1.0$
Guard(300)	$W \le W_{\mathrm{FCFS}} + 300$ s	0.705/192.5	0.738/209.7	0.478/229.1
Guard(600)	$W \le W_{\mathrm{FCFS}} + 600$ s	0.740/492.2	0.837/507.8	0.801/523.7
Guard(1200)	$W \le W_{\mathrm{FCFS}} + 1200$ s	0.750/996.7	0.850/1096.1	0.844/1110.4
CBF-L	start by the reservation made at arrival	0.000/0.0	0.000/0.0	0.000/0.0
Slack-L(G)	start by the plan at arrival $+ G$	0.000/0.4	0.000/0.9	0.000/2.0
CBF-P	none	0.020/64.9	0.012/71.0	0.008/121.1
Slack-P(G)	none	0.557/1333.3	0.696/3089.4	0.656/5369.9

Ordering by score inside windows of arrivals limits overtaking by position, and it keeps less of the benefit than the guard. We use Block(b) of Table 3 in place of the closest scheduler for judging queues [73]. We ran it on the first overlay with blocks of 8, 32, 128 and 512 jobs and no selection step, and at every load, the largest block does best. It closes at most 0.417 of the gap, at the lightest load, while Guard(300) closes at least 0.486 at every load on the same overlay. Near saturation, a block large enough to help also lets long jobs wait longer than the guard does, with a largest excess of 444.5 s against 219.6 s. The repeated selection of Section 7.3 put Block on the guard's footing, with the online score and the same harm limit on the fifteen validation cells. There, we also ran blocks of 2048 and 8192 jobs, beyond the largest block above. Wider blocks close more of the gap on the worst cell, but they also harm short jobs more. Block(8192) closes 0.524 of the gap there, with a worst harm of 747.1 s, above the harm limit at every bound. The best feasible block closes 0.031 at $G = 600$ s and 0.087 at $G = 1200$ s. The guard closes 0.591 and 0.703 there.

Under deliberately corrupted scores, the guard still keeps every job within its bound, but it no longer keeps the tail short. We reversed the scores, drew them at random, or scored the longest 1% of jobs as the shortest (the inverted-tail score of Table 3). The unguarded policy then delays some job by more than 6000 s beyond FCFS at the busiest load. The guarded $\mathrm{p99}_{\mathrm{dl}}$ is 546.5, 261.9 and 543.6 s under the three corrupted scores, against FCFS's 273.50 s. Over these three scores and the five protocols of Section 8.1, the largest excess of Guard(600) over FCFS is at most 534.8 s at the busiest load. A guarded run can thus have a worse percentile than FCFS, by at most G (Supplementary Table S8).

8.5. The Bounds on the Traces

No job exceeds any of the bounds of Section 6 in any run, and on one server the worst job comes within 1% of the guard's bound. Table 12 pairs each result with the experiment that tests it. A test on the replayed logs shows how much of a bound a real load uses, and a test on constructed instances shows that a constant cannot be lowered.

Table 12. The results of Section 6 and the experiments that test them. "Trace" marks a test on the replayed logs, and "construction" a test on built instances.

Result	What It States	Test	Outcome
Lemma 1, Theorem 1	$k(W_P[i] - W_{\mathrm{FCFS}}[i])$ equals $\mathrm{In}_i - \mathrm{Out}_i$ up to $2(k-1)L$, and exactly at $k = 1$	trace: In_i and Out_i recorded for every job, 27 policies at three loads (Supplementary Section S3.13)	no violation in 1.43 billion checks; residual at most 0.518 of $2(k-1)L$; $(\mathrm{In}_i - \mathrm{Out}_i)/k$ explains $R^2 = 0.983$ to 0.9996 of the excess
Theorem 2	the constant $2(k-1)L$ cannot be lowered	construction: a geometric cascade (Supplementary Section S8.2)	the cascade approaches the constant; the replayed loads stay near $(k-1)L$
Corollaries 1 and 2	a budget of work suffices for a guarantee, and every guarantee implies one	trace: Skip(G), which counts overtakes, beside Guard(G) at the same G (Supplementary Figure S2)	no violation; the count that gives the same G leaves the queue close to FCFS at the two higher loads
Theorem 3, Corollary 3	$W_P[i] < W_{\mathrm{FCFS}}[i] + G$ for every job, around any base policy and any scores	trace: the bound asserted on every job of every guarded run, corrupted scores included	no violation in over 10 billion checks; the worst job uses up to 0.998 of G on one server and 0.66 to 0.93 at the main loads (Figure 5)
Proposition 1	Timeout(θ) keeps $W_P[i] < W_{\mathrm{FCFS}}[i] + \theta + (3 - 2/k)L$	trace: Timeout checked job by job on the fifteen main cells (Table 10)	no violation in any run
Proposition 2	$W_P[i] < (\sigma + B_{\max})/k + (2 - 1/k)L$	trace: 304 runs of FCFS, the work budgets and Skip	no violation; the worst wait uses 0.999 of the bound at $k = 1$ and 0.881 to 0.965 at the three loads
Proposition 3	with reports up to δ late, the bound grows by $L + \delta$	construction (Supplementary Section S4.12); trace with δ from 0.1 to 300 s (Section 8.9)	no violation in 1665 guarded runs; up to $\delta = 30$ s no job exceeds even the bound of Theorem 3
Proposition 4, tightness of Theorem 3	a budget rule of the family either makes the guard FCFS around every base policy or cannot guarantee a bound below L for every base policy, and both constants of the guard bound are exact	constructions (Supplementary Sections S4.8, S8.3 and S8.4); trace on one server	the worst job reaches 0.995 of the bound at $k = 1$, so the additive term is reached in practice
Theorem 4	on long and short jobs, a guarantee G caps the share of SJF's reduction of the mean wait at about $(kG + (3k-2)L)/(nc)$	construction: exhaustive check over 1161 settings (Supplementary Sections S8.5 and S8.6); illustration on the traces: gap closed against G (Supplementary Figure S2)	the guard around exact SJF meets the cap up to $2(3k-2)L/(nc)$; on the traces the gap closed rises with G, as the cap does

In every run of a bounded policy, the simulation checks the bound on every job. A violation would therefore expose an error in the code rather than in the theorem. The 315 runs reported here, one per policy and cell, make 5,554,949,400 such checks. With the single-server runs and an independently written second simulator, the count exceeds ten billion, and none of the checks finds a violation. We checked Proposition 1 job by job on the same overlays, and Proposition 2 on runs of FCFS, the work budgets and Skip. Neither check found a violation. Proposition 3 was checked on constructed instances (Supplementary Section S4.12) and on the cells of Table 9 with late reports (Section 8.9). In these runs

the ratio of the realised worst excess to the bound falls as k grows. It is 0.9950 in the single-server runs, where the guard under the worst corrupted score reaches it, and 0.8850 at $k = 8$ (Supplementary Sections S3.6 and S3.16). The capacity scan of Figure 5(a) varies k on the arrivals of the busiest load, and on one server its ratio reaches 0.998.

The bound exceeds the observed worst case by much only where the load leaves little to protect. Figure 5 compares the bound G with the largest excess that any job attains, across the number of servers, the time limit and the load. The ratio of the two remains below one in every setting, as Theorem 3 requires. On one server the worst job comes within 1% of the bound, so the additive term is attained in practice. With the arrivals held fixed, additional servers reduce the load, and at 20 servers the worst excess is at most 0.41 of G, because few jobs then need protection. At a fixed G, a longer time limit increases the additive term $(3 - 2/k)L$ and reduces the cap, and the worst excess decreases as L grows. For $L \geq 60$ s the running times stay those of $L = 60$ s, because they are already cut at 60 s, and so does the predictor. The scan therefore changes the guard's parameters and not the jobs.

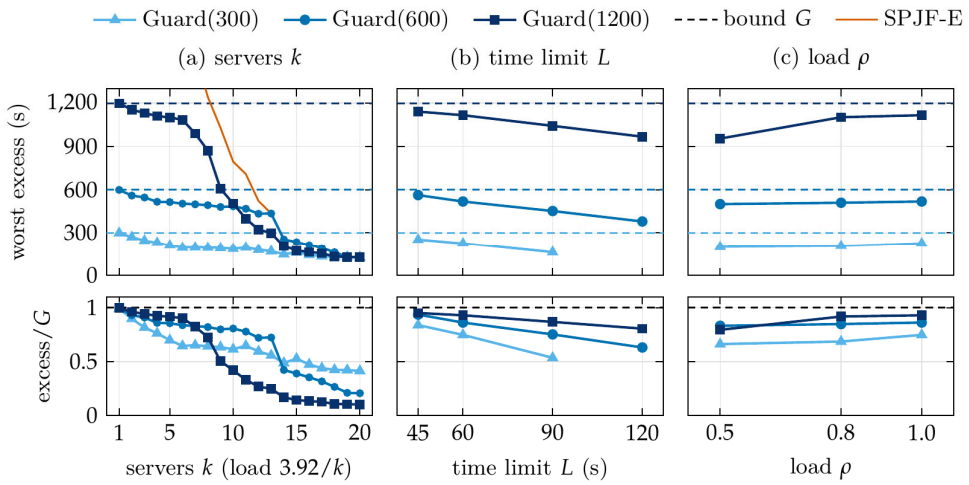


Figure 5. The guaranteed bound against the worst excess observed. Upper row: the bound G of Guard(G) (dashed, in the colour of each guard) and the largest excess over FCFS that any job reaches under it (solid), in seconds; lower row: their ratio, which the guarantee keeps below 1. (**a**) Servers $k = 1$ to 20 on the arrivals of $\rho = 1.0$, so the load falls as $3.92/k$; original protocol, archived scores (stored SPJF-E predictions, not refitted; the capacity scan of Supplementary Table S19). The unguarded SPJF-E (thin line) has a worst excess below $G = 1200$, 600 and 300 s from $k = 9$, 12 and 14; from there on, the distance to the bound shows that the load needs little guarding, not that the bound is loose. (**b**) Time limit L with G held fixed, so the cap $B_{\max} = k(G - (3 - 2/k)L)$ shrinks as the additive term $(3 - 2/k)L$ grows. The running times are already cut at 60 s, so above 60 s they and the predictor stay those of $L = 60$ s, and only the guard's parameters change. The excess shown is the worst over the three loads, under the online protocol. At $L = 120$ s the promise $G = 300$ s leaves no budget, so Guard(300) is FCFS and is not drawn. At $L = 45$ s jobs are cut at 45 s, so FCFS and the reference change. (**c**) The three loads of Table 9 ($k = 7.8$, 5 and 4) at $L = 60$ s, online protocol (Table 7). The ratio ranges over 0.104–0.998 in (**a**), 0.536–0.951 in (**b**) and 0.663–0.930 in (**c**).

The absolute bound of Proposition 2 comes close to the realised waits, but a bound quoted in advance needs a margin on the burst. At one server every FCFS wait equals its fluid backlog to the microsecond, as the proposition predicts, and the largest wait nearly reaches the bound. At the three loads the worst wait also uses most of the bound, under FCFS and under the work budgets alike (Table 12). A bound quoted in advance needs a

forecast of the burst σ. The burst of the validation terms is too low for the evaluation terms in 10 of 15 cells, and a bound quoted from it is broken in 18 of 60 runs, by up to 349.8 s. Such a bound therefore needs a margin—here, a factor of 1.36 on σ—or an admission control that keeps the arrivals within the quoted burst.

8.6. Budget Shapes on One to Three Servers

On one to three servers, the age-relative shape is the best of the three in every setting, and the constant default suffices at the loosest bound. The shape parameters were selected on cells with four to eight servers, and on one server, the shape selected at $G = 1200$ s does worse than the others (Supplementary Section S3.14). We therefore ran every shape on one, two and three servers. Each setting has five overlays at three loads, with utilisations of 0.5, 0.8 and 0.95 in the busy hours. The one-server overlays come from the pipeline of the single-server runs of Supplementary Section S3.6, run at these three loads. The two- and three-server overlays come from the same pipeline with the work scaled by k (Supplementary Section S13.1). The shape points are those selected for the main tables. Besides the constant, age-relative and queue-length shapes, we ran Fixed(G) and the constant budget $B_{\max}/2$ of Section 5.4, which we call the default. Supplementary Table S35 gives every setting. In 4.7 billion checks, no job exceeds its bound.

Among the three shapes, the age-relative one closes the largest share of the gap in all 27 settings, and its harm is at most $G/2$ in 26 of them. On one server, the queue-length shape selected at $G = 1200$ s closes 0.70 to 0.76 of the gap, against 0.85 to 0.87 for the age-relative shape. Fixed(G) closes about as much as the age-relative shape, but its harm exceeds $G/2$ in 21 of the 27 settings. Fixed(G) differs from the shapes only in its budget, the whole cap $B_{\max}$ for every job (Table 3). At the same bound, the smaller budgets of the shapes therefore keep their harm lower.

Before the runs, we defined the default as adequate in a setting if it falls short of the best shape by at most 0.05 of the gap and its harm is at most $G/2$. At $G = 1200$ s the default is adequate in all nine settings, with a shortfall of at most 0.047. At tighter bounds it falls short as the load rises, by up to 0.142 at 600 s and 0.291 at 300 s. The shortfall depends more on the bound and the load than on the number of servers. On one to three servers of this log, these results give a rule that needs no grid search. An operator sets $B_{\max}$ from G and uses the default, which sufficed at $G = 20L$ in every setting. For tighter bounds, the age-relative shape with B_0 and η as selected here was the better choice on one, two and three servers.

8.7. Where the Method Applies

The guard is useful where the FCFS tail is long against the time limit and the costs are concentrated, which holds for the main log and the cloud-function log. Table 13 places the main log beside the other workloads. Excesses are in units of each log's L, the unit in which the guarantee is written. For the grading logs L is the time limit, and for the others it is the analysis cap of Section 7.1. Two screens show where the guard is useful. The first compares the FCFS tail with the guarantee's additive term $(3 - 2/k)L$, which is close to $3L$. If the FCFS p99 is below this term, the guarantee is coarser than the waits themselves. Only CodeBench and the cloud-function log pass the first screen. The compute farm is the extreme case, with an FCFS p99 of 18.7 h against $L = 493{,}857$ s. The second screen measures how concentrated the costs are, which is the opportunity for reordering. In the two CI pools, the longest 1% of jobs carry only a small share of the work, and ordering by true cost makes the p99 worse in three of the four windows that we examined. Two logs pass both screens, CodeBench and the cloud-function log. On them, the guard closes about half or more of the gap and holds the largest excess below $4L$ at a bound of about $5L$. We drew up the screens on these same logs. The 900 s class of LPC-EGEE passes the first

screen only at one server, the smallest capacity of its search, with an FCFS p99 of $13.8L$. With as many servers as the class itself used at once, its FCFS p99 is zero (Supplementary Section S6.6). At one server, the guard closes 0.569 of the gap in the mean wait. Its gap in the busy-hour p99, -0.103 $[-0.542, 0.560]$, is not resolved (Supplementary Table S23). Passing both screens does not guarantee a gain, as the next subsection shows.

Table 13. The workloads under one format. The rows are applicability diagnostics, each under its own arrival, capacity and cost construction, and not replications of one experiment. They carry no sampling intervals. Tail share: fraction of executed work carried by the longest 1% of jobs. AUROC: heavy label from information available at arrival. The p99 is that of the waits before deadlines on the two grading logs and that of all waits on the other logs. Screen 1: whether the FCFS p99 exceeds $(3 - 2/k)L$. Excess is in units of the L of each log. The guard column uses $G = 5L$. Every log's grid contains this bound: 300 s on CodeBench, $\{2,5,15\}L$ on the cloud-function log and the compute farm, and $\{3,5,10\}L$ on the CI pools (ACcoding: footnote a). A negative gap closed means that the p99 of the policy is worse than that of FCFS. CodeBench row: original protocol. The unguarded and the guarded columns can use different base scores (footnote c). Supplementary Section S6.7 describes the CI capacities and adds a log from the grid computing cluster of the Laboratoire de Physique Corpusculaire in the Enabling Grids for E-science in Europe project (LPC-EGEE).

	Tail		**FCFS**	**Screen**	**Gap Closed**	**Unguarded**	**Guard at $G \approx 5L$**
Log and load	share	AUROC	p99/L	1	SPJF [c]	max excess	gap closed/max excess
CodeBench, $\rho \approx 1.0$	0.375	0.936	4.56	yes	0.916	98.4L	0.478/3.82L
ACcoding, drawn arrivals, $\rho = 0.8$ [a]	0.122	0.927	0.050	no	0.980	20L	0.974/2.21L
Azure Functions, $\rho = 1.0$	0.443	0.985	4.22	yes	0.858	10.07L	0.710/2.52L
CI pool A, $k = 156$ of 173	0.034	0.651	1.78	no	n/a [b]	11.53L	0.735 (mean) [b]/2.64L
CI pool B, $k = 96$ of 105	0.052	0.997	0.373	no	0.958	0.34L	0.958/0.34L
Intel Netbatch, $\rho = 0.95$	0.372	0.890	0.136	no	-4.111	0.85L	-4.346/0.83L

[a] ACcoding arrivals are drawn, so this row describes the ordering and not waiting on that platform. They differ from the arrivals of the sealed ACcoding run of Section 8.2, which places submissions around contest start times. Its guard uses the largest budget tested, $B = 10L$ of work, which at $k = 5$ gives a bound of about $4.6L$. Its tail share is simulated with $L = 30.5$ s (0.275 on measured cost at $L = 30.0$ s, the figure of Section 1). [b] On CI pool A, SJF is itself worse than FCFS at the p99, so no p99 gap is defined. The guard's entry is the gap in mean wait and is not comparable with the other rows. [c] The unguarded column orders by SPJF-E on CodeBench and CI pool A and by SPJF-log on the other four logs. The guard wraps SPJF-E, except on ACcoding and CI pool B, where it wraps SPJF-log.

8.8. Capacity, Physical Runs and Feedback

The benefit of the guard depends on capacity. It shrinks once the busiest hours exceed full load, first at the tighter bounds. We ran every k from 1 to 20 on the five overlays, under the original protocol (Supplementary Table S19). These runs use the archived scores, that is, the stored SPJF-E predictions of the development pipeline, without refitting. Within each run the capacity is fixed, so this scan is a sensitivity analysis and not a study of capacity planning. At $k = 1$, Guard(600) makes $\mathrm{p99}_{\mathrm{dl}}$ worse by 493.79 s. It improves the p99 on all five overlays only for $k = 4$ to 12. Figure 6 draws this scan against the offered load ρ. The scan's Guard(1200) is the point that the first run of the selection rule chose (Supplementary Section S5). It differs from the final one only in starting from $B_0 = 0$. Below $\rho = 1$, Guard(600) and Guard(1200) close 0.66 to 0.85 of the gap, while Guard(300) falls to 0.47 as the load nears one. Above $\rho = 1$, the benefit of each guard decreases to about zero, first at the tighter bounds. Unguarded SPJF-E continues to improve the p99 up to $\rho = 2$, at a harm of up to 1010 s. However, on one server, at $\rho = 3.9$, it does worse than FCFS, with a harm of 159,132 s.

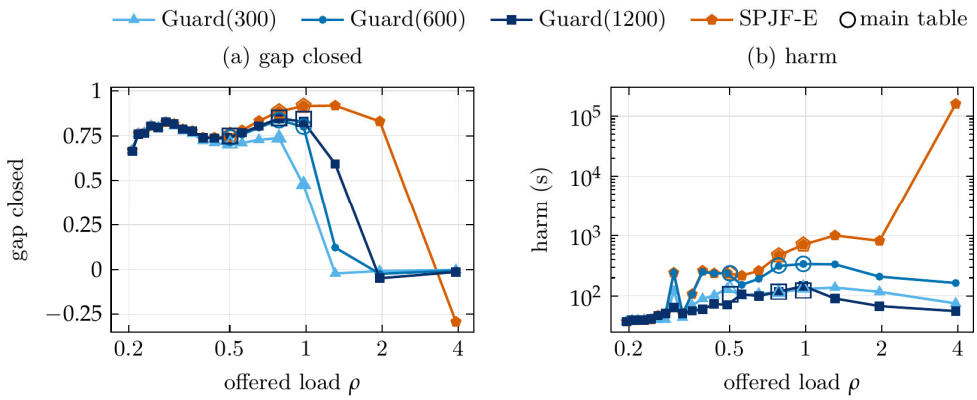


Figure 6. Where the guard helps on the main log. (**a**) Gap closed and (**b**) harm, the largest excess among jobs that FCFS starts within 1 s, of Guard(G) at three promises and of unguarded SPJF-E, against the offered load ρ. Lines: the capacity scan of Supplementary Table S19, with $k = 1$ to 20 servers on fixed arrivals, so that $\rho = 3.92/k$; original protocol, archived scores. For the lines, gap closed is the ratio of the differences in $\mathrm{p99}_{\mathrm{dl}}$, each averaged over the overlays, and Guard(1200) starts from $B_0 = 0$. Rings: the cells of Table 9 ($k = 7.8,\ 5$ and 4), whose gap closed is the mean of the per-overlay ratios, with the guard as selected. Gap closed is undefined at $k = 20$, where FCFS and SJF both have $\mathrm{p99}_{\mathrm{dl}} = 0$. The vertical distance from a guard's line to SPJF-E's in (**a**) is the gap that its promise costs.

The guard keeps its bound when it runs as a separate service on one machine. We ran FCFS, SPJF-E and Guard(300) in turn on two worker processes, which hold each job for its recorded duration in real time. In each run, the workers executed the 2151 jobs of the 300 s development window with the most work, and all 2151 dispatch decisions followed the policy's rule. We compared each job with FCFS on the same run's measured arrivals and running times. Every job stayed within the bound of Guard(300), with a largest excess of 222.21 s. In a physical run, the bound must be widened by the time that workers sat idle during dispatch. This time was at most 1.127 s here, and no job came close to needing it. Both processes ran on one machine, so this run tests the dispatch rule and not network delay. The window held 6.864 times the work that the two workers could do in it. Each policy started empty and drained its own queue. In this burst, the guarded p99 of 1783.65 s was no better than the 1763.30 s of a separately timed FCFS run. The mean wait, however, was 480.39 s under the guard against 1337.05 s under FCFS. The separate runs had different measured arrivals and running times, so these comparisons are descriptive (Supplementary Section S6.2). A second physical test used the two-worker back end of a course platform, where twenty scripted accounts submitted 100 jobs per cell in a deadline-shaped burst with $G = 400$ s. Under unguarded predicted order, seven jobs exceeded the bound. In the four guarded cells, no job exceeded it, and the worst excess was $0.514G$ (Supplementary Section S6.9).

Restoring part of the users' feedback changes the size of the waits. At the two higher loads, it does not change the share of the gap that the guard closes. The simulations replay arrivals at their recorded times. A scheduler is therefore neither rewarded with work that it would attract nor punished for users that it would drive away [110,111]. The logs record one part of that feedback, the time between one result and the same student's next submission. In a second replay, each submission is released only after its predecessor's result plus that recorded delay. We ran it on all five overlays. This replay shortens $\mathrm{p99}_{\mathrm{dl}}$ by a factor of 1.3 to 3.8. The reason is that when a queue builds up, the submissions that would deepen it are held back. At the two higher loads, Guard(600) then closes about as much

of the gap as with the recorded arrival times. At the lightest load, it closes more (Supplementary Table S21). We bootstrapped the change in the gap closed over weeks on the first overlay. It is $+0.123$ [0.043, 0.193] at the lightest load and $+0.010$ [−0.037, 0.063] at the busiest. It keeps every job within its bound (Supplementary Section S6.5).

The log records when each result was written and how long each run took, and we take the reconstructed start of the run as the submission time (Section 7.1). It therefore cannot show how long a student would wait or when a student would give up. We added to this replay a stated model of abandonment, which we fixed before the first run. Each attempt has an exponential patience with mean 60, 300 or 1800 s. An attempt that is still waiting when its patience runs out leaves the queue. Its student then resubmits it with probability 0, 0.5 or 1, at most three times. On the fifteen development cells, FCFS has the highest $\mathrm{p99}_{\mathrm{dl}}$ in every combination, and SJF has the lowest. The p99 counts only the attempts that were served, and it falls as patience shortens because the longest waits end in withdrawals. As patience shortens, the gap closed by SPJF-E and Guard(600) narrows from 0.81–0.85 with infinite patience to 0.66–0.73 at a mean of 60 s. Consider the combinations in which at least 1000 jobs are withdrawn under SPJF-E. There, the number withdrawn under SPJF-E or the guard is 0.21 to 0.71 times the number under FCFS. Under the guard, it is 0.98 to 1.08 times the number under SPJF-E. Under this patience model, size-aware order thus leads to fewer withdrawn jobs than FCFS, and the guard leads to about as many as SPJF-E. The bound refers to FCFS run on the realised arrivals of the served attempts. Against that reference, every job of the 450 guarded runs met the bound. A second reference is FCFS that loses its own users under the same patience. This reference runs on a different input, so Theorem 3 does not cover it. Against that reference, 8484 jobs in 144 of the 450 guarded runs exceeded the bound, all of them in runs with resubmission. We claim no bound against it (Supplementary Section S13.3).

8.9. The Cost of Late Completion Reports

Late completion reports cost little on this load. Up to a delay of 30 s, no job exceeds even the bound for prompt reports. Proposition 3 bounds the guard when completion reports reach the scheduler up to δ late. To see what such delays cost on the real load, we replayed the 15 cells of Table 9 with each charge applied after a delay, for δ from 0.1 to 300 s. We ran the three versions of the counter of Section 6.8. They are one shared counter, a counter that each server updates with its own completion before it takes its next job, and a separate view for each server. Delays were either constant or uniform on $[0, \delta]$. At $\delta = 0$ the simulator with delays reproduces every wait of the main simulator on the first overlay. On 30,000 random instances, it also agrees job by job with a separate reference implementation. Table 14 reports the shared counter with constant delays, the version with the largest effect.

Late charges make the guard fire later, so the schedule moves toward SPJF-E. Under Guard(300), which fires most often, the gap closed therefore rises with δ at every load. Under the two larger bounds, it changes by at most 0.012 below 300 s, in either direction (Table 14). The cost appears in the worst case, where the largest excess grows but stays within the $L + \delta$ of Proposition 3. At $\delta = 60$ s three jobs exceed the bound of Theorem 3 under Guard(300), and one job does under Guard(600). At 300 s, many jobs do. In every run, every job meets the bound of Proposition 3. When each server counts its own completion before it takes its next job, no job exceeds the bound of Theorem 3 at any δ, as the last sentence of Proposition 3 states. On these cells the largest excess and the harm then equal those of $\delta = 0$, and $\mathrm{p99}_{\mathrm{dl}}$ differs from that of $\delta = 0$ by at most 0.006 s. Separate views per server give smaller increases than the shared counter, and below $\delta = 300$ s no job exceeds the bound of Theorem 3. For comparison, completion reports in the physical run of Section 8.8 reached the scheduler 0.17 to 0.21 ms after the completion, on average.

https://doi.org/10.3390/math14214020

Table 14. Late completion reports, one shared counter, constant delay δ (seconds). For each bound, the gap closed at the three loads (means over the five overlays), and the largest excess over all 15 cells beside the bound $G + L + \delta$ of Proposition 3. At $\delta = 0$, the runs reproduce Table 9. Original protocol.

	Guard(300)		Guard(600)		Guard(1200)	
δ	Gap Closed	Excess/Bound	Gap Closed	Excess/Bound	Gap Closed	Excess/Bound
0	0.705/0.738/0.478	229/360	0.740/0.837/0.801	524/660	0.750/0.850/0.844	1110/1260
0.1	0.705/0.739/0.479	229/360.1	0.741/0.837/0.801	523/660.1	0.750/0.851/0.844	1115/1260.1
1	0.706/0.740/0.478	230/361	0.740/0.836/0.800	531/661	0.750/0.850/0.845	1120/1261
5	0.707/0.747/0.489	256/365	0.741/0.839/0.804	548/665	0.746/0.852/0.845	1136/1265
30	0.719/0.761/0.514	283/390	0.741/0.840/0.811	572/690	0.738/0.848/0.846	1178/1290
60	0.727/0.778/0.544	305/420	0.745/0.842/0.813	605/720	0.742/0.846/0.845	1180/1320
300	0.738/0.827/0.721	532/660	0.748/0.850/0.832	825/960	0.745/0.854/0.861	1384/1560

8.10. Overhead of the Accounts

The guard's accounts add about 0.4 μs per job to the time of SPJF-E in our simulation kernel (Table 15). We measured the processor time of the simulating thread over whole runs on the first overlay, on an Intel Core Ultra 9 275HX with no other experiment running. The guard roughly doubles the time per event. When we timed the guard's operations alone, one job cost 310 to 373 ns with 100 to 100,000 waiting jobs, and 1.47 μs with one million. In the physical run above, a guarded dispatch decision took 0.574 ms on average, against 0.044 ms for SPJF-E. That run used a plain Python version of the guard, which checks every waiting job and writes an audit record for each. Its time therefore grows with the queue, and its 99th percentile was 1.13 ms. The kernel keeps the accounts in the trees of Section 5.4, so the figures above give the cost of the accounts themselves. Both physical figures are small beside the 7.6 to 8.2 ms by which job releases missed their recorded times in that run.

Table 15. Processor time of the simulation kernel on the first overlay (n = 17,634,760 jobs, three events per job). Each entry is the range, over the three loads, of the median over five repeated runs.

Policy	ns per Event	Extra ns per Job over SPJF-E	Processor s per Run
FCFS	114.6–124.3	5–11	6.06–6.58
SPJF-E	111.6–120.8	0	5.91–6.39
Guard(300)	246.3–254.0	391–404	13.03–13.44
Guard(600)	241.6–251.0	390–396	12.78–13.28
Guard(1200)	242.5–259.0	385–415	12.83–13.70

Replicas of the scheduler that share the arrival order and the dispatch history need no state beyond the completion reports (Section 6.8). Each report carries one job number and one cost, so sending the reports to r such replicas takes r messages per completion. This count does not include keeping their dispatch histories the same. A replica that receives a report late behaves as a scheduler whose charges arrive late. Proposition 3 bounds this case.

9. Limitations

The guarantee compares each job with FCFS on the same sequence of arrivals. When users react to their waits, by resubmitting or by leaving, the sequence itself depends on the policy. The guarantee then says nothing about the waits under the sequence that FCFS would have induced. Supplementary Section S6.3 constructs feedback under which a scheduling advantage disappears or reverses. The guarantee bounds the wait of a job that

is stopped at the time limit, but it says nothing about the result that such a job loses. Preemption is outside the model, and work conservation cannot be dropped (Section 6.9).

Every additive constant in Section 6 is a multiple of the time limit, apart from the report delay δ and the arrival burst σ. The guard's constant $(3 - 2/k)L$ cannot be improved. Consider any budget rule of the guard's family that ever lets the base policy reorder. No such rule can guarantee less than L around every base policy (Proposition 4). Task assignment by guessing size stops long jobs and restarts them in a second tier [112]. On a shared pool, this does not escape the floor either. The stopped work is lost and not overtaken, and the second tier is again non-preemptive in blocks of L (Supplementary Section S7.5). Charging a cap that is known at arrival lowers the constant by exactly L, but the floor remains. Where the FCFS tail is already below about $3L$, the bound is coarser than the waits that it bounds.

The guarantee bounds each job, so it also bounds each percentile. Every percentile of the waits under the guard is at most the same percentile under FCFS plus G. The guarantee does not ensure that a percentile improves.

The bounds for late reports need a known bound δ on the delay of each report. Lost reports are covered only when each server applies its own report before it takes its next job (Proposition 3). The model counts a dispatch as instantaneous. Suppose that a server must wait for its next job. The bound of a job then grows by two terms (Section 6.8): the job's own delay between dispatch and start, and the idle time accumulated since the pool last held no unfinished work, divided by k. Replicas of the scheduler agree on the guard's decisions only when they share the arrival order and the dispatch history. The analysis does not cover how they keep these the same.

One question remains open: Suppose that an overtaker started in the same phase as the job is charged only the work that it executes while the job waits (Supplementary Section S4.3). The lower constant of the identity is then known only to lie in $[(k-1)L, 2(k-1)L]$.

Algorithm 1 is also not the least restrictive guard with its promise. For $k \geq 2$, a wrapper that reserves L for each overtaker still in service keeps the same G. At every decision, whatever the history and the proposed job, it overrides the base policy only if Algorithm 1 would, and at some decisions, it does not (Section 6.5 and Supplementary Section S7.6).

10. Conclusions

A service that runs many users' jobs on a few shared servers can order its queue by predicted running time. It can still guarantee every job a wait at most G seconds longer than in arrival order on the same arrivals, whatever the predictions are. Earlier results on scheduling with predictions bound a mean, a competitive ratio or a difference between clients (Section 2). This guarantee instead holds for each job, and it follows from one identity. A job's extra wait relative to FCFS equals the net running time of its overtakers divided by the number of servers, within $(2 - 2/k)L$. Limiting that overtaking work is therefore sufficient for a per-job guarantee and, up to an additive $O(kL)$, necessary.

The identity holds for every policy that never interrupts a job and never leaves a server idle while a job waits. Any such scheduler that limits the work passing each waiting job therefore bounds every job's extra wait, whatever it ranks by (Corollary 1). From its own log, a scheduler can also estimate each job's extra wait over FCFS without a replay of FCFS. The net overtaking work divided by k is within $(2 - 2/k)L$ of that extra wait.

The guard enforces such a limit with the exact constant $(3 - 2/k)L$, and completion reports that arrive up to δ late add at most $L + \delta$. A maximum waiting time gives a guarantee of the same form, but it charges a job for all the work ahead of it. Near full load on the main log, it kept much less of the benefit at bounds of $5L$ and $10L$. At $20L$ the two kept about the same share.

https://doi.org/10.3390/math14214020

For the grading system with which this paper opened, the answer to a student whose submission sat behind a looping one is now a number. SJF reduces the 99th-percentile wait before deadlines relative to FCFS. On three terms held out until the method was frozen, a guard with $G = 600$ s keeps 0.894 of that reduction. No job exceeds its bound there. Resampling whole class-terms instead of weeks gives this share a wide interval (Section 8.2). On the main log, the benefit shrank once the busiest hours exceeded full load.

The scores come from gradient-boosted trees on each student's and each task's history. We chose this model on queueing outcomes and not on prediction accuracy. On one test term, a heterogeneous graph network predicts about as well. On new tasks, its gain comes from the neural model of each job's own features and, for our graph design, not from the graph's edges.

An operator can set the guard without a grid search. The operator states a bound G above $(3 - 2/k)L$ and sets the cap $B_{\max} = k(G - (3 - 2/k)L)$ by Corollary 3. The constant budget $B_{\max}/2$ needs no data and keeps every wait less than $(G + (3 - 2/k)L)/2$ above its FCFS wait. On one to three servers of the main log, this default came within 0.05 of the best shape's gap closed whenever $G = 20L$ (Section 8.6). For tighter bounds, the age-relative shape did better on these servers with the parameters of Section 5.4, which we selected on four to eight servers. The shape parameters never change the guarantee, so a poor choice costs part of the gain but not the bound.

Supplementary Materials: The following supporting information can be downloaded at https://www.mdpi.com/article/10.3390/math14214020/s1: Table S1: Predictors on the non-judging logs, by feature group; Table S2: Predictors on the two grading logs under one protocol; Table S3: Share of work carried by the longest jobs, and the provenance of L; Table S4: Experimental configuration and selection protocol; Table S5: Selected budget points and per-shape grid winners; Table S6: The guard at three bounds, with the comparators; Table S7: The age-relative and queue-length winners; Table S8: Deliberately corrupted scores; Table S9: The residual of Theorem 1 on nine cells; Table S10: The single-server configuration; Table S11: Definition of linear aging; Table S12: Comparison with the aging baseline; Table S13: Predictor AUROC with user-block intervals; Table S14: Alternative ranking scores; Table S15: Trace-construction sensitivities; Table S16: Unguarded policies; Table S17: Candidates and feasible points of the selection; Table S18: Whole weeks against whole class-terms as the sampling unit; Table S19: Capacity sweep; Table S20: Physical timed-work runs; Table S21: Replay with recorded think times; Table S22: A grid log by walltime class; Table S23: The grid log's 900 s class at its own limit; Table S24: The grid-log rows of the cross-log comparison; Table S25: Six cells on a course platform's back end; Table S26: Scope of the nearest per-job guarantees; Table S27: Sealed terms: ranking scores; Table S28: Sealed terms: the guard and the comparators; Table S29: Sealed terms: corrupted scores; Table S30: Sealed terms: a single server; Table S31: The main policies under the online replay; Table S32: Paired differences on both clocks; Table S33: The guard against a maximum waiting time under the online replay; Table S34: Results of Section 6 and the Lean theorems that prove them; Table S35: Budget shapes on one to three servers; Table S36: The sealed part of a second platform under the frozen protocol; Table S37: User abandonment in the think-time replay; Table S38: Whole weeks against whole class-terms under the online replay; Table S39: The same resampling on the sealed terms; Figure S1: Gap–harm trade-offs of the budget shapes; Figure S2: What each budget shape keeps as the bound is relaxed; Figure S3: Original-score reference and corrupted-score controls. The Supplementary Materials also give the proofs deferred from Section 6 (Sections S4, S7 and S8), extended related work (Section S9), the sealed terms (Section S10), the comparisons under the online replay (Section S11), the machine-checked proofs (Section S12) and the experiments added in the revision (Section S13). References [113–131] are cited only in the Supplementary Materials, whose own reference list numbers them differently.

Author Contributions: Conceptualisation, Y.G. and Z.L.; methodology, Y.G.; software, Z.L.; validation, Z.L. and C.H.; formal analysis, C.H.; investigation, Z.L.; resources, Y.S.; data curation, Y.G. and

Z.L.; writing—original draft preparation, Y.G. and Z.L.; writing—review and editing, C.H. and Y.S.; visualisation, Z.L.; supervision, C.H.; project administration, Y.S.; funding acquisition, Y.G. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Nanjing University of Posts and Telecommunications, grant number XC2025021.

Institutional Review Board Statement: Not applicable. This study used previously published, pseudonymised logs and involved no human participants and no animals.

Informed Consent Statement: Not applicable.

Data Availability Statement: The public source datasets used here are available at the locations cited, and we redistribute none of them. The CodeBench judging logs are published by the Universidade Federal do Amazonas [21]. The ACcoding dataset is published with its data descriptor [22], and the OULAD with its own [105]. The Azure Functions 2021 invocation trace is published in the Azure Public Dataset repository [5], which asks that [6] be cited. The Intel Netbatch logs are published in the Parallel Workloads Archive [25], which asks that the logs be acknowledged to their depositors and suggests citing [26]. The LPC-EGEE log is also published in the Parallel Workloads Archive [108], with acknowledgment requested for its data provider. We collected the CI trace of the two Firefox hardware worker pools from Mozilla's public, unauthenticated Taskcluster and Treeherder application programming interfaces [23,24]. We do not redistribute it, but the released code includes the collection scripts that rebuild it from those interfaces. The publisher distributes the Azure invocation trace under a Creative Commons Attribution licence [5]. Public availability therefore does not imply an absence of licence conditions. We used no restricted access agreement, and we processed no personal data beyond the pseudonymous user identifiers that the published judging datasets already carry. We archived the simulator, the predictors, the guard implementation, the configurations, the Lean proofs and the earlier development analyses at Zenodo under the concept DOI https://doi.org/10.5281/zenodo.23035899. The version used here is 1.1.0-revision, https://doi.org/10.5281/zenodo.23643805, and the same files are in the repository https://github.com/oscar030406/spjf-guard (accessed on 31 October 2026). The code is released under the MIT licence. Supplementary Section S5 distinguishes the tables reproduced by the package from the earlier development evidence.

Acknowledgments: The authors thank Ohad Shai, Edi Shmueli and Nir Antebi of Intel, who provided the Intel Netbatch logs, and Emmanuel Medernach, who provided the LPC-EGEE log, through the Parallel Workloads Archive. During the preparation of this study and manuscript, the authors used Claude Code (Anthropic, version 2.1, with the Claude Opus 5.5 and Claude Fable 5.1 models) and Codex CLI (OpenAI, version 0.163, with the GPT-5.6-Sol and GPT-6-Astra models) for the purposes described in Section 7.4. The authors have reviewed and edited the output and take full responsibility for the content of this publication.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AUROC	Area under the receiver operating characteristic curve
CBF	Conservative backfilling
CI	Continuous integration
EASY	Extensible Argonne Scheduling sYstem
ERE	Ensino Remoto Emergencial (emergency remote teaching)
FCFS	First-come first-served
GPS	Generalized processor sharing
GRU	Gated recurrent unit

HGNN	Heterogeneous graph neural network
LPC-EGEE	Laboratoire de Physique Corpusculaire, Enabling Grids for E-science in Europe
OULAD	Open University Learning Analytics Dataset
p99	99th percentile
$p99_{dl}$	99th percentile of the waits of the jobs that arrive in the 24 h before a deadline
PFCFS	Preemptive first-come first-served
PGPS	Packet-by-packet generalized processor sharing
R-GCN	Relational graph convolutional network
RMSE	Root mean squared error
SJF	Shortest job first
SOAP	Schedule ordered by age-based priority
SPJF	Shortest predicted job first
SPJF-E	Shortest predicted job first on the expected-cost score
SPJF-log	Shortest predicted job first on the log-scale score
SRPT	Shortest remaining processing time

References

1. Wasik, S.; Antczak, M.; Badura, J.; Laskowski, A.; Sternal, T. A Survey on Online Judge Systems and Their Applications. *ACM Comput. Surv.* **2019**, *51*, 3. https://doi.org/10.1145/3143560.
2. König, C.J.; Kleinmann, M. Deadline Rush: A Time Management Phenomenon and Its Mathematical Description. *J. Psychol.* **2005**, *139*, 33–45. https://doi.org/10.3200/JRLP.139.1.33-45.
3. Alfi, V.; Parisi, G.; Pietronero, L. Conference registration: How people react to a deadline. *Nat. Phys.* **2007**, *3*, 746. https://doi.org/10.1038/nphys761.
4. Alfi, V.; Gabrielli, A.; Pietronero, L. How people react to a deadline: Time distribution of conference registrations and fee payments. *Open Phys.* **2009**, *7*, 483–489. https://doi.org/10.2478/s11534-009-0059-z.
5. Microsoft Azure. Azure Functions Invocation Trace 2021. Available online: https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsInvocationTrace2021.md (accessed on 19 September 2026).
6. Zhang, Y.; Goiri, Í.; Chaudhry, G.I.; Fonseca, R.; Elnikety, S.; Delimitrou, C.; Bianchini, R. Faster and Cheaper Serverless Computing on Harvested Resources. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), Virtual, 26–29 October 2021; pp. 724–739. https://doi.org/10.1145/3477132.3483580.
7. Peveler, M.; Maicus, E.; Cutler, B. Comparing Jailed Sandboxes vs Containers Within an Autograding System. In Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19), Minneapolis, MN, USA, 27 February–2 March 2019; pp. 139–145. https://doi.org/10.1145/3287324.3287507.
8. DOMjudge Manual: Overview. Available online: https://www.domjudge.org/docs/manual/main/overview.html (accessed on 19 September 2026).
9. Lykouris, T.; Vassilvitskii, S. Competitive Caching with Machine Learned Advice. *J. ACM* **2021**, *68*, 24. https://doi.org/10.1145/3447579.
10. Purohit, M.; Svitkina, Z.; Kumar, R. Improving Online Algorithms via ML Predictions. In Proceedings of the Advances in Neural Information Processing Systems 31 (NeurIPS 2018), Montréal, QC, Canada, 3–8 December 2018. Available online: https://papers.nips.cc/paper_files/paper/2018/hash/73a427badebe0e32caa2e1fc7530b7f3-Abstract.html (accessed on 31 October 2026).
11. Im, S.; Kumar, R.; Montazer Qaem, M.; Purohit, M. Non-Clairvoyant Scheduling with Predictions. In Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '21), Virtual, 6–8 July 2021; pp. 285–294. https://doi.org/10.1145/3409964.3461790.
12. Lindermayr, A.; Megow, N. Permutation Predictions for Non-Clairvoyant Scheduling. In Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '22), Philadelphia, PA, USA, 11–14 July 2022; pp. 357–368. https://doi.org/10.1145/3490148.3538579.
13. Scully, Z.; Grosof, I.; Mitzenmacher, M. Uniform Bounds for Scheduling with Job Size Estimates. In *Proceedings of the 13th Innovations in Theoretical Computer Science Conference (ITCS 2022), Berkeley, CA, USA, 31 January–3 February 2022*; LIPIcs; Schloss Dagstuhl–Leibniz-Zentrum für Informatik: Wadern, Germany, 2022; Volume 215, pp. 114:1–114:30. https://doi.org/10.4230/LIPIcs.ITCS.2022.114.
14. Mitzenmacher, M.; Shahout, R. Queueing, Predictions, and Large Language Models: Challenges and Open Problems. *Stoch. Syst.* **2025**, *15*, 195–219. https://doi.org/10.1287/stsy.2025.0106.

15. Zhao, T.; Xia, C.; Chang, X.; Li, C.; Li, W.; Zomaya, A. Competitive Fair Scheduling with Predictions. In Proceedings of the Thirteenth International Conference on Learning Representations (ICLR 2025), Singapore, 24–28 April 2025; pp. 66548–66585.
16. Yu, G.; Scully, Z. Strongly Tail-Optimal Scheduling in the Light-Tailed M/G/1. *Proc. ACM Meas. Anal. Comput. Syst.* **2024**, *8*, 27. https://doi.org/10.1145/3656011.
17. Yu, G.; Harlev, A.; Adakroy, R.; Scully, Z. A Tale of Two Traffics: Optimizing Tail Latency in the Light-Tailed M/G/k. *Proc. ACM Meas. Anal. Comput. Syst.* **2025**, *9*, 46. https://doi.org/10.1145/3771561.
18. Ke, G.; Meng, Q.; Finley, T.; Wang, T.; Chen, W.; Ma, W.; Ye, Q.; Liu, T.Y. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In Proceedings of the Advances in Neural Information Processing Systems 30 (NIPS 2017), Long Beach, CA, USA, 4–9 December 2017; pp. 3146–3154.
19. Cho, K.; van Merriënboer, B.; Gulcehre, C.; Bahdanau, D.; Bougares, F.; Schwenk, H.; Bengio, Y. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), Doha, Qatar, 25–29 October 2014; pp. 1724–1734. https://doi.org/10.3115/v1/D14-1179.
20. Schlichtkrull, M.; Kipf, T.N.; Bloem, P.; van den Berg, R.; Titov, I.; Welling, M. Modeling Relational Data with Graph Convolutional Networks. In *Proceedings of the 15th European Semantic Web Conference (ESWC 2018)*; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2018; Volume 10843, pp. 593–607. https://doi.org/10.1007/978-3-319-93417-4_38.
21. Universidade Federal do Amazonas, Instituto de Computação. CodeBench Dataset, Version 1.81. Available online: https://codebench.icomp.ufam.edu.br/dataset/ (accessed on 19 September 2026).
22. Chen, K.; Huang, F.; Liu, Z.; Yu, H.; Meng, L.; Mo, S.; Zhang, L.; Song, Y. ACcoding: A Graph-Based Dataset for Online Judge Programming. *Sci. Data* **2024**, *11*, 548. https://doi.org/10.1038/s41597-024-03392-z. Correction in *Sci. Data* **2024**, *11*, 833
23. Mozilla. Taskcluster Documentation: Queue Service Reference. Available online: https://docs.taskcluster.net/docs/reference/platform/queue (accessed on 19 September 2026).
24. Mozilla. Treeherder REST API Documentation. Available online: https://treeherder.mozilla.org/docs/ (accessed on 19 September 2026).
25. Parallel Workloads Archive. The Intel Netbatch Logs. Available online: https://www.cs.huji.ac.il/labs/parallel/workload/l_intel_netbatch/ (accessed on 19 September 2026).
26. Shai, O.; Shmueli, E.; Feitelson, D.G. Heuristics for Resource Matching in Intel's Compute Farm. In *Proceedings of the Job Scheduling Strategies for Parallel Processing (JSSPP 2013)*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2014; Volume 8429, pp. 116–135. https://doi.org/10.1007/978-3-662-43779-7_7.
27. Lattanzi, S.; Lavastida, T.; Moseley, B.; Vassilvitskii, S. Online Scheduling via Learned Weights. In Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms (SODA), Salt Lake City, UT, USA, 5–8 January 2020; pp. 1859–1877. https://doi.org/10.1137/1.9781611975994.114.
28. Mitzenmacher, M. Scheduling with Predictions and the Price of Misprediction. In *Proceedings of the 11th Innovations in Theoretical Computer Science Conference (ITCS 2020)*; LIPIcs; Schloss Dagstuhl—Leibniz-Zentrum für Informatik: Wadern, Germany, 2020; Volume 151, pp. 14:1–14:18. https://doi.org/10.4230/LIPIcs.ITCS.2020.14.
29. Azar, Y.; Leonardi, S.; Touitou, N. Flow Time Scheduling with Uncertain Processing Time. In Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing (STOC), Virtual, 21–25 June 2021; pp. 1070–1080. https://doi.org/10.1145/3406325.3451023.
30. Azar, Y.; Leonardi, S.; Touitou, N. Distortion-Oblivious Algorithms for Minimizing Flow Time. In Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), Virtual, 9–12 January 2022; pp. 252–274. https://doi.org/10.1137/1.9781611977073.13.
31. Dong, J.; Ibrahim, R. Shortest-Job-First Scheduling in Many-Server Queues with Impatient Customers and Noisy Service-Time Estimates. *Oper. Res.* **2025**, *73*, 3139–3155. https://doi.org/10.1287/opre.2022.310.
32. Shahout, R.; Mitzenmacher, M. SkipPredict: When to Invest in Predictions for Scheduling. In Proceedings of the Advances in Neural Information Processing Systems 37 (NeurIPS 2024), Vancouver, BC, Canada, 10–15 December 2024; pp. 105663–105701. https://doi.org/10.52202/079017-3352.
33. Benomar, Z.; Perchet, V. Non-clairvoyant Scheduling with Partial Predictions. In *Proceedings of the 41st International Conference on Machine Learning (ICML 2024)*; Proceedings of Machine Learning Research; PMLR: New York, NY, USA, 2024; Volume 235, pp. 3506–3538.

34. Gupta, A.; Kumar, A.; Panigrahi, D.; Wang, Z. An Optimal Online Algorithm for Robust Flow Time Scheduling. In *Proceedings of the 2026 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*; SIAM: Philadelphia, PA, USA, 2026; pp. 1214–1238. https://doi.org/10.1137/1.9781611978971.47.
35. Zhao, T.; Zomaya, A.Y. On Smoothness Bounds for Non-Clairvoyant Scheduling with Predictions. In Proceedings of the Fourteenth International Conference on Learning Representations (ICLR 2026), Rio de Janeiro, Brazil, 23–27 April 2026; pp. 140116–140147.
36. Moseley, B.; Newman, H.; Pruhs, K.; Zhou, R. Robust Gittins for Stochastic Scheduling. *ACM SIGMETRICS Perform. Eval. Rev.* **2025**, *53*, 166–168. https://doi.org/10.1145/3744970.3727315. Full version: arXiv:2504.10743.
37. Ramakrishna, S.; Harlev, A.; Scully, Z. Empirical Gittins for Data-Driven M/G/1 Scheduling with Arbitrary Job Size Distributions. *Proc. ACM Meas. Anal. Comput. Syst.* **2026**, *10*, 9. https://doi.org/10.1145/3788091.
38. Merlis, N.; Richard, H.; Sentenac, F.; Odic, C.; Molina, M.; Perchet, V. On Preemption and Learning in Stochastic Scheduling. In *Proceedings of the 40th International Conference on Machine Learning (ICML 2023)*; Proceedings of Machine Learning Research; PMLR: New York, NY, USA, 2023; Volume 202, pp. 24478–24516.
39. Blue, M.; Im, S.; Lindermayr, A. Learning-Augmented Online Scheduling with Parsimonious Preemption. *arXiv* **2026**, https://doi.org/10.48550/arXiv.2605.23255.
40. Akbari-Moghaddam, M.; Down, D.G. SEH: Size Estimate Hedging Scheduling of Queues. *ACM Trans. Model. Comput. Simul.* **2023**, *33*, 14. https://doi.org/10.1145/3580491.
41. Shahout, R.; Malach, E.; Liu, C.; Jiang, W.; Yu, M.; Mitzenmacher, M. Don't Stop Me Now: Embedding Based Scheduling for LLMs. In Proceedings of the Thirteenth International Conference on Learning Representations (ICLR 2025), Singapore, 24–28 April 2025; pp. 63345–63368.
42. Sheng, Y.; Cao, S.; Li, D.; Zhu, B.; Li, Z.; Zhuo, D.; Gonzalez, J.E.; Stoica, I. Fairness in Serving Large Language Models. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*; USENIX Association: Berkeley, CA, USA, 2024; pp. 965–988.
43. Jaillet, P.; Jiang, J.; Mellou, K.; Molinaro, M.; Podimata, C.; Zhou, Z. Online Scheduling for LLM Inference with KV Cache Constraints. *arXiv* **2025**, https://doi.org/10.48550/arXiv.2502.07115.
44. Chen, Z.; Ye, Y.; Zhou, Z. Adaptively Robust LLM Inference Optimization under Prediction Uncertainty. *arXiv* **2025**, https://doi.org/10.48550/arXiv.2508.14544.
45. Rutten, D.; Christianson, N.; Mukherjee, D.; Wierman, A. Smoothed Online Optimization with Unreliable Predictions. *Proc. ACM Meas. Anal. Comput. Syst.* **2023**, *7*, 12. https://doi.org/10.1145/3579442.
46. Yang, J.; Li, P.; Li, T.; Wierman, A.; Ren, S. Anytime-Competitive Reinforcement Learning with Policy Prior. In Proceedings of the Advances in Neural Information Processing Systems 36 (NeurIPS 2023), New Orleans, LA, USA, 10–16 December 2023; pp. 77852–77866. https://doi.org/10.52202/075280-3401.
47. Li, P.; Christianson, N.; Yang, J.; Wierman, A.; Ren, S. Learning for Sustainable Online Scheduling with Competitive Fairness Guarantees. In Proceedings of the 16th ACM International Conference on Future and Sustainable Energy Systems (e-Energy '25), Rotterdam, The Netherlands, 17–20 June 2025; pp. 63–79. https://doi.org/10.1145/3679240.3734615.
48. Addanki, V.; Pacut, M.; Schmid, S. Credence: Augmenting Datacenter Switch Buffer Sharing with ML Predictions. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*; USENIX Association: Berkeley, CA, USA, 2024; pp. 613–634.
49. Saxena, D.; Chen, J.; Yadalam, S.; Ro, Y.; Dwivedula, R.; Campbell, E.H.; Akella, A.; Rossbach, C.J.; Swift, M. How I Learned to Stop Worrying and Love Learned OS Policies. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '25)*; ACM: New York, NY, USA, 2025; pp. 1–7. https://doi.org/10.1145/3713082.3730384.
50. Xie, R.; Grosof, I.; Scully, Z. Heavy-Traffic Optimal Size- and State-Aware Dispatching. *Proc. ACM Meas. Anal. Comput. Syst.* **2024**, *8*, 9. https://doi.org/10.1145/3639035.
51. Grosof, I.; Scully, Z.; Harchol-Balter, M. Load Balancing Guardrails: Keeping Your Heavy Traffic on the Road to Low Response Times. *Proc. ACM Meas. Anal. Comput. Syst.* **2019**, *3*, 42. https://doi.org/10.1145/3341617.3326157.
52. Agrawal, K.; Baruah, S.; Bender, M.A.; Marchetti-Spaccamela, A. The Safe and Effective Use of Low-Assurance Predictions in Safety-Critical Systems. In *Proceedings of the 35th Euromicro Conference on Real-Time Systems (ECRTS 2023)*; LIPIcs; Schloss Dagstuhl—Leibniz-Zentrum für Informatik: Wadern, Germany, 2023; Volume 262, pp. 3:1–3:19. https://doi.org/10.4230/LIPIcs.ECRTS.2023.3.
53. Schwiegelshohn, U.; Yahyapour, R. Fairness in Parallel Job Scheduling. *J. Sched.* **2000**, *3*, 297–320. https://doi.org/10.1002/1099-1425(200009/10)3:5<297::AID-JOS50>3.0.CO;2-D.

54. Sabin, G.; Kochhar, G.; Sadayappan, P. Job Fairness in Non-Preemptive Job Scheduling. In *Proceedings of the 2004 International Conference on Parallel Processing (ICPP 2004)*; IEEE Computer Society: Washington, DC, USA, 2004; pp. 186–194. https://doi.org/10.1109/ICPP.2004.1327920.
55. Sabin, G.; Sadayappan, P. Unfairness Metrics for Space-Sharing Parallel Job Schedulers. In *Proceedings of the Job Scheduling Strategies for Parallel Processing (JSSPP 2005)*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2005; Volume 3834, pp. 238–256. https://doi.org/10.1007/11605300_12.
56. Mu'alem, A.W.; Feitelson, D.G. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Trans. Parallel Distrib. Syst.* **2001**, *12*, 529–543. https://doi.org/10.1109/71.932708.
57. Lindsay, A.M.; Galloway-Carson, M.; Johnson, C.R.; Bunde, D.P.; Leung, V.J. Backfilling with guarantees made as jobs arrive. *Concurr. Comput. Pract. Exp.* **2013**, *25*, 513–523. https://doi.org/10.1002/cpe.2860.
58. Talby, D.; Feitelson, D.G. Supporting Priorities and Improving Utilization of the IBM SP Scheduler Using Slack-Based Backfilling. In Proceedings of the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP 1999), San Juan, Puerto Rico, 12–16 April 1999; pp. 513–517. https://doi.org/10.1109/IPPS.1999.760525.
59. Lifka, D.A. The ANL/IBM SP Scheduling System. In *Proceedings of the Job Scheduling Strategies for Parallel Processing (IPPS '95 Workshop)*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 1995; Volume 949, pp. 295–303. https://doi.org/10.1007/3-540-60153-8_35.
60. Yuan, Y.; Yang, G.; Wu, Y.; Zheng, W. PV-EASY: A Strict Fairness Guaranteed and Prediction Enabled Scheduler in Parallel Job Scheduling. In Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC '10), Chicago, IL, USA, 21–25 June 2010; pp. 240–251. https://doi.org/10.1145/1851476.1851505.
61. Grosof, I.; Yang, K.; Scully, Z.; Harchol-Balter, M. Nudge: Stochastically Improving upon FCFS. *Proc. ACM Meas. Anal. Comput. Syst.* **2021**, *5*, 21. https://doi.org/10.1145/3460088.
62. Van Houdt, B. On the Stochastic and Asymptotic Improvement of First-Come First-Served and Nudge Scheduling. *Proc. ACM Meas. Anal. Comput. Syst.* **2022**, *6*, 50. https://doi.org/10.1145/3570610.
63. Charlet, N.; Van Houdt, B. Tail Optimality and Performance Analysis of the Nudge*(M) Scheduling Algorithm. *Oper. Res.* **2026**, *74*, 2161–2178. https://doi.org/10.1287/opre.2024.1061.
64. Grosof, I.; Harchol-Balter, M.; Scheller-Wolf, A. WCFS: A new framework for analyzing multiserver systems. *Queueing Syst.* **2022**, *102*, 143–174. https://doi.org/10.1007/s11134-022-09848-6.
65. Demers, A.; Keshav, S.; Shenker, S. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of the Symposium on Communications Architectures & Protocols (SIGCOMM '89)*; ACM: New York, NY, USA, 1989; pp. 1–12. https://doi.org/10.1145/75246.75248.
66. Parekh, A.K.; Gallager, R.G. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Trans. Netw.* **1993**, *1*, 344–357. https://doi.org/10.1109/90.234856.
67. Bennett, J.C.R.; Zhang, H. WF ^{2}Q: Worst-case Fair Weighted Fair Queueing. In Proceedings of the IEEE INFOCOM '96, San Francisco, CA, USA, 24–28 March 1996; Volume 1, pp. 120–128. https://doi.org/10.1109/INFCOM.1996.497885.
68. Stoica, I.; Abdel-Wahab, H. *Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation*; Technical Report TR-95-22; Revised 26 January 1996; Department of Computer Science, Old Dominion University: Norfolk, VA, USA, 1995.
69. Graham, R.L. Bounds on Multiprocessing Timing Anomalies. *SIAM J. Appl. Math.* **1969**, *17*, 416–429. https://doi.org/10.1137/0117039.
70. Tsafrir, D.; Etsion, Y.; Feitelson, D.G. Backfilling Using System-Generated Predictions Rather than User Runtime Estimates. *IEEE Trans. Parallel Distrib. Syst.* **2007**, *18*, 789–803. https://doi.org/10.1109/TPDS.2007.70606.
71. Gaussier, E.; Glesser, D.; Reis, V.; Trystram, D. Improving backfilling by using machine learning to predict running times. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15), Austin, TX, USA, 15–20 November 2015; pp. 1–10. https://doi.org/10.1145/2807591.2807646.
72. Gaussier, E.; Lelong, J.; Reis, V.; Trystram, D. Online Tuning of EASY-Backfilling Using Queue Reordering Policies. *IEEE Trans. Parallel Distrib. Syst.* **2018**, *29*, 2304–2316. https://doi.org/10.1109/TPDS.2018.2820699.
73. Zhang, E.; Wu, F.; Lu, X. An Intelligent Scheduling System for Large-Scale Online Judging. In *Proceedings of the Computer Science and Education. Computer Science and Technology (ICCSE 2023)*; Communications in Computer and Information Science; Springer Nature: Singapore, 2024; Volume 2023, pp. 265–279. https://doi.org/10.1007/978-981-97-0730-0_24.

74. Fu, Y.; Zhu, S.; Su, R.; Qiao, A.; Stoica, I.; Zhang, H. Efficient LLM Scheduling by Learning to Rank. In Proceedings of the Advances in Neural Information Processing Systems 37 (NeurIPS 2024), Vancouver, BC, Canada, 10–15 December 2024; pp. 59006–59029. https://doi.org/10.52202/079017-1882.
75. Li, Y.; Chen, Y.; Chen, J.; Choukse, E.; Qiu, H.; Suh, G.E.; Fonseca, R.; Scully, Z.; Gupta, U. Beyond Prediction: Tail-Aware Scheduling for LLM Inference. In *Proceedings of the 43rd International Conference on Machine Learning (ICML 2026), Seoul, Republic of Korea, 6–11 July 2026*; Proceedings of Machine Learning Research; PMLR: New York, NY, USA, 2026; Volume 306, pp. 66188–66203.
76. Wu, B.; Zhong, Y.; Zhang, Z.; Liu, S.; Liu, F.; Sun, Y.; Huang, G.; Liu, X.; Jin, X. FastServe: Iteration-Level Preemptive Scheduling for Large Language Model Inference. In Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26), Renton, WA, USA, 4–6 May 2026; pp. 57–74.
77. Luo, M.; Shi, X.; Cai, C.; Zhang, T.; Wong, J.; Wang, Y.; Wang, C.; Huang, Y.; Chen, Z.; Gonzalez, J.E.; et al. Autellix: An Efficient Serving Engine for LLM Agents as General Programs. *arXiv* **2025**, https://doi.org/10.48550/arXiv.2502.13965.
78. Yang, M.; Wang, G.; Luo, M.; Liu, Y.; Chen, C.; Zhao, H.; Feng, Y.; Chen, Q.; Guo, M. Justitia: Fair and Efficient Scheduling of Task-Parallel LLM Agents with Selective Pampering. *arXiv* **2025**, arXiv:2510.17015.
79. Fu, Y.; Liu, L.; Wang, H.; Cheng, Y.; Chen, S. SFS: Smart OS Scheduling for Serverless Functions. In *Proceedings of the SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*; IEEE: Piscataway, NJ, USA, 2022; pp. 1–16. https://doi.org/10.1109/SC41404.2022.00047.
80. Fu, Y.; Shi, R.; Wang, H.; Chen, S.; Cheng, Y. ALPS: An Adaptive Learning, Priority OS Scheduler for Serverless Functions. In *Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC 24)*; USENIX Association: Berkeley, CA, USA, 2024; pp. 19–36.
81. Massey, W.A.; Whitt, W. Peak congestion in multi-server service systems with slowly varying arrival rates. *Queueing Syst.* **1997**, *25*, 157–172. https://doi.org/10.1023/A:1019156418862.
82. Jennings, O.B.; Mandelbaum, A.; Massey, W.A.; Whitt, W. Server Staffing to Meet Time-Varying Demand. *Manag. Sci.* **1996**, *42*, 1383–1394. https://doi.org/10.1287/mnsc.42.10.1383.
83. Robinson, J.; Ranjan, R.; Hu, W.; Huang, K.; Han, J.; Dobles, A.; Fey, M.; Lenssen, J.E.; Yuan, Y.; Zhang, Z.; et al. RelBench: A Benchmark for Deep Learning on Relational Databases. In Proceedings of the Advances in Neural Information Processing Systems 37, Datasets and Benchmarks Track (NeurIPS 2024), Vancouver, BC, Canada, 10–15 December 2024; pp. 21330–21341. https://doi.org/10.52202/079017-0672.
84. Grinsztajn, L.; Oyallon, E.; Varoquaux, G. Why Do Tree-Based Models Still Outperform Deep Learning on Typical Tabular Data? In Proceedings of the Advances in Neural Information Processing Systems 35 (NeurIPS 2022), New Orleans, LA, USA, 28 November–9 December 2022; pp. 507–520. https://doi.org/10.52202/068431-0037.
85. Harchol-Balter, M. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*; Cambridge University Press: Cambridge, UK, 2013. https://doi.org/10.1017/CBO9781139226424.
86. Scully, Z.; Harchol-Balter, M.; Scheller-Wolf, A. SOAP: One Clean Analysis of All Age-Based Scheduling Policies. *Proc. ACM Meas. Anal. Comput. Syst.* **2018**, *2*, 16. https://doi.org/10.1145/3179419.
87. Smith, W.E. Various optimizers for single-stage production. *Nav. Res. Logist. Q.* **1956**, *3*, 59–66. https://doi.org/10.1002/nav.3800030106.
88. Jørgensen, B. Exponential Dispersion Models. *J. R. Stat. Soc. Ser. B* **1987**, *49*, 127–145. https://doi.org/10.1111/j.2517-6161.1987.tb01685.x.
89. Yang, Y.; Qian, W.; Zou, H. Insurance Premium Prediction via Gradient Tree-Boosted Tweedie Compound Poisson Models. *J. Bus. Econ. Stat.* **2018**, *36*, 456–470. https://doi.org/10.1080/07350015.2016.1200981.
90. Fenwick, P.M. A new data structure for cumulative frequency tables. *Softw. Pract. Exp.* **1994**, *24*, 327–336. https://doi.org/10.1002/spe.4380240306.
91. de Moura, L.; Ullrich, S. The Lean 4 Theorem Prover and Programming Language. In *Proceedings of the Automated Deduction—CADE 28*; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2021; Volume 12699, pp. 625–635. https://doi.org/10.1007/978-3-030-79876-5_37.
92. The mathlib Community. The Lean Mathematical Library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2020)*; ACM: New York, NY, USA, 2020; pp. 367–381. https://doi.org/10.1145/3372885.3373824.
93. Schrage, L.E.; Miller, L.W. The Queue M/G/1 with the Shortest Remaining Processing Time Discipline. *Oper. Res.* **1966**, *14*, 670–684. https://doi.org/10.1287/opre.14.4.670.

94. Grosof, I.; Scully, Z.; Harchol-Balter, M. SRPT for multiserver systems. *Perform. Eval.* **2018**, *127–128*, 154–175. https://doi.org/10.1016/j.peva.2018.10.001.
95. Daley, D.J. Certain optimality properties of the first-come first-served discipline for G/G/s queues. *Stoch. Processes Their Appl.* **1987**, *25*, 301–308. https://doi.org/10.1016/0304-4149(87)90208-0.
96. Kleinrock, L. A conservation law for a wide class of queueing disciplines. *Nav. Res. Logist. Q.* **1965**, *12*, 181–192. https://doi.org/10.1002/nav.3800120206.
97. Bartsch, B.; Bolch, G. A conservation law for G/G/m queueing systems. *Acta Inform.* **1978**, *10*, 105–109. https://doi.org/10.1007/BF00260928.
98. Green, T.C.; Stidham, S., Jr. Sample-path conservation laws, with applications to scheduling queues and fluid systems. *Queueing Syst.* **2000**, *36*, 175–199. https://doi.org/10.1023/A:1019183220080.
99. El-Taha, M. A general workload conservation law with applications to queueing systems. *Queueing Syst.* **2017**, *85*, 361–381. https://doi.org/10.1007/s11134-017-9515-4.
100. Bender, M.A.; Chakrabarti, S.; Muthukrishnan, S. Flow and Stretch Metrics for Scheduling Continuous Job Streams. In Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), San Francisco, CA, USA, 25–27 January 1998; pp. 270–279.
101. Mastrolilli, M. Notes on Max Flow Time Minimization with Controllable Processing Times. *Computing* **2003**, *71*, 375–386. https://doi.org/10.1007/s00607-003-0029-z.
102. Cruz, R.L. A Calculus for Network Delay, Part I: Network Elements in Isolation. *IEEE Trans. Inf. Theory* **1991**, *37*, 114–131. https://doi.org/10.1109/18.61109.
103. Le Boudec, J.Y.; Thiran, P. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*; Lecture Notes in Computer Science; Springer: Berlin, Heidelberg, 2001; Volume 2050. https://doi.org/10.1007/3-540-45318-0.
104. Wolff, R.W. Upper bounds on work in system for multichannel queues. *J. Appl. Probab.* **1987**, *24*, 547–551. https://doi.org/10.2307/3214279.
105. Kuzilek, J.; Hlosta, M.; Zdrahal, Z. Open University Learning Analytics dataset. *Sci. Data* **2017**, *4*, 170171. https://doi.org/10.1038/sdata.2017.171.
106. Universidade Federal do Amazonas, Instituto de Computação. CodeBench. Available online: https://codebench.icomp.ufam.edu.br/ (accessed on 23 September 2026).
107. Smith, D.R.; Whitt, W. Resource Sharing for Efficiency in Traffic Systems. *Bell Syst. Tech. J.* **1981**, *60*, 39–55. https://doi.org/10.1002/j.1538-7305.1981.tb00221.x.
108. Parallel Workloads Archive. The LPC Log. Available online: https://www.cs.huji.ac.il/labs/parallel/workload/l_lpc/ (accessed on 23 September 2026).
109. Harchol-Balter, M.; Schroeder, B.; Bansal, N.; Agrawal, M. Size-based scheduling to improve web performance. *ACM Trans. Comput. Syst.* **2003**, *21*, 207–233. https://doi.org/10.1145/762483.762486.
110. Feitelson, D.G. Resampling with Feedback: A New Paradigm of Using Workload Data for Performance Evaluation. In *Job Scheduling Strategies for Parallel Processing (JSSPP 2021)*; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2021; Volume 12985, pp. 3–32. https://doi.org/10.1007/978-3-030-88224-2_1.
111. Shmueli, E.; Feitelson, D.G. On Simulation and Design of Parallel-Systems Schedulers: Are We Doing the Right Thing? *IEEE Trans. Parallel Distrib. Syst.* **2009**, *20*, 983–996. https://doi.org/10.1109/TPDS.2008.152.
112. Harchol-Balter, M. Task assignment with unknown duration. *J. ACM* **2002**, *49*, 260–288. https://doi.org/10.1145/506147.506154.
113. Bansal, N.; Harchol-Balter, M. Analysis of SRPT Scheduling: Investigating Unfairness. In Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, Cambridge, MA, USA, 16–20 June 2001; pp. 279–290. https://doi.org/10.1145/378420.378792.
114. Bennett, J.C.R.; Benson, K.; Charny, A.; Courtney, W.F.; Le Boudec, J.Y. Delay Jitter Bounds and Packet Scale Rate Guarantee for Expedited Forwarding. *IEEE/ACM Trans. Netw.* **2002**, *10*, 529–540. https://doi.org/10.1109/TNET.2002.801404.
115. Federgruen, A.; Groenevelt, H. M/G/c Queueing Systems with Multiple Customer Classes: Characterization and Control of Achievable Performance Under Nonpreemptive Priority Rules. *Manag. Sci.* **1988**, *34*, 1121–1138. https://doi.org/10.1287/mnsc.34.9.1121.
116. Gu, J.; Ranjan, R.; Kanatsoulis, C.; Tang, H.; Jurkovic, M.; Hudovernik, V.; Znidar, M.; Chaturvedi, P.; Shroff, P.; Li, F.; et al. RelBench v2: A Large-Scale Benchmark and Repository for Relational Data. *arXiv* **2026**, https://doi.org/10.48550/arXiv.2602.12606.

117. Jeffay, K.; Stanat, D.F.; Martel, C.U. On Non-Preemptive Scheduling of Periodic and Sporadic Tasks. In Proceedings of the 12th IEEE Real-Time Systems Symposium (RTSS 1991), San Antonio, TX, USA, 4–6 December 1991; pp. 129–139. https://doi.org/10.1109/REAL.1991.160366.
118. Kiefer, J.; Wolfowitz, J. On the theory of queues with many servers. *Trans. Am. Math. Soc.* **1955**, *78*, 1–18. https://doi.org/10.1090/S0002-9947-1955-0066587-3.
119. Kim, W.S.; Lim, D.E. Analysis of overtaking in M/M/c queues. *Comput. Ind. Eng.* **2016**, *101*, 177–183. https://doi.org/10.1016/j.cie.2016.09.005.
120. Kleinrock, L. A delay dependent queue discipline. *Nav. Res. Logist. Q.* **1964**, *11*, 329–341. https://doi.org/10.1002/nav.3800110306.
121. Li, N.; Stanford, D.A. Multi-server accumulating priority queues with heterogeneous servers. *Eur. J. Oper. Res.* **2016**, *252*, 866–878. https://doi.org/10.1016/j.ejor.2016.02.010.
122. Parekh, A.K.; Gallager, R.G. A generalized processor sharing approach to flow control in integrated services networks: The multiple node case. *IEEE/ACM Trans. Netw.* **1994**, *2*, 137–150. https://doi.org/10.1109/90.298432.
123. Rajbhandary, A.; Bunde, D.P.; Leung, V.J. Variations of Conservative Backfilling to Improve Fairness. In *Proceedings of the Job Scheduling Strategies for Parallel Processing (JSSPP 2013)*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2014; Volume 8429, pp. 177–191. https://doi.org/10.1007/978-3-662-43779-7_10.
124. Shahrad, M.; Fonseca, R.; Goiri, Í.; Chaudhry, G.; Batum, P.; Cooke, J.; Laureano, E.; Tresness, C.; Russinovich, M.; Bianchini, R. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC 20)*; USENIX Association: Berkeley, CA, USA, 2020; pp. 205–218.
125. Sharif, A.B.; Stanford, D.A.; Taylor, P.; Ziedins, I. A multi-class multi-server accumulating priority queue with application to health care. *Oper. Res. Health Care* **2014**, *3*, 73–79. https://doi.org/10.1016/j.orhc.2014.01.002.
126. Stanford, D.A.; Taylor, P.; Ziedins, I. Waiting time distributions in the accumulating priority queue. *Queueing Syst.* **2014**, *77*, 297–330. https://doi.org/10.1007/s11134-013-9382-6.
127. Verma, A.; Pedrosa, L.; Korupolu, M.; Oppenheimer, D.; Tune, E.; Wilkes, J. Large-scale cluster management at Google with Borg. In Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15), Bordeaux, France, 21–24 April 2015; pp. 1–17. https://doi.org/10.1145/2741948.2741964.
128. Whitt, W. The amount of overtaking in a network of queues. *Networks* **1984**, *14*, 411–426. https://doi.org/10.1002/net.3230140305.
129. Wierman, A.; Harchol-Balter, M. Classifying scheduling policies with respect to unfairness in an M/GI/1. In Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, San Diego, CA, USA, 10–14 June 2003; pp. 238–249. https://doi.org/10.1145/781027.781057.
130. Wierman, A. Fairness and scheduling in single server queues. *Surv. Oper. Res. Manag. Sci.* **2011**, *16*, 39–48. https://doi.org/10.1016/j.sorms.2010.07.002.
131. Yuan, Y.; Wu, Y.; Zheng, W.; Li, K. Guarantee Strict Fairness and Utilize Prediction Better in Parallel Job Scheduling. *IEEE Trans. Parallel Distrib. Syst.* **2014**, *25*, 971–981. https://doi.org/10.1109/TPDS.2013.88.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.