QGym: Scalable Simulation and Benchmarking of Queuing Network Controllers

Haozhe Chen[1,*] **Ang Li**[1,*] **Ethan Che**[2,*]
Tianyi Peng[2] **Jing Dong**[2] **Hongseok Namkoong**[2]

[1]Columbia University [2]Columbia Business School
{haozhe.chen, al4263}@columbia.edu
{eche25, tianyi.peng, jing.dong, namkoong}@gsb.columbia.edu

Abstract

Queuing network control determines the allocation of scarce resources to manage congestion, a fundamental problem in manufacturing, communications, and healthcare. Compared to standard RL problems, queueing problems are distinguished by unique challenges: i) a system operating in continuous time, ii) high stochasticity, and iii) long horizons over which the system can become unstable (exploding delays). To spur methodological progress tackling these challenges, we present an open-sourced queueing simulation framework, QGym, that benchmark queueing policies across realistic problem instances. Our modular framework allows the researchers to build on our initial instances, which provide a wide range of environments including parallel servers, criss-cross, tandem, and re-entrant networks, as well as a realistically calibrated hospital queuing system. QGym makes it easy to compare multiple policies, including both model-free RL methods and classical queuing policies. Our testbed complements the traditional focus on evaluating algorithms based on mathematical guarantees in idealized settings, and significantly expands the scope of empirical benchmarking in prior work. QGym code is open-sourced at `https://github.com/namkoong-lab/QGym`.

1 Introduction

Queuing network control is a fundamental control problem in managing congestion in job-processing systems, such as semiconductor manufacturing fabrication plants, communications networks, cloud computing facilities, call centers, healthcare delivery systems, ride-sharing platforms, and limit-order books [44, 23, 37, 2, 4, 7, 14]. In a typical queuing system, jobs arrive at random intervals, wait in queues until an available server can service them, and then either leave the system or move on to another queue for further processing. The stochastic workload is the defining challenge in queueing: the variability inherent in real-world systems makes the time it takes to process jobs random. To manage performance objectives such as minimizing processing delays, balancing workloads, and improving overall service quality and efficiency, a good controller must dynamically allocate resources accounting for future stochasticity. The ability to plan is especially crucial for systems that experience varying levels of congestion over time.

Routing/scheduling control (i.e., matching jobs with servers) in industrial-scale systems is challenging due to several factors. First, queueing networks can be large and complex, with many different job classes and server types (see, e.g., [23, 4]). The processing speeds can depend on the server and job

[*]Equal contribution.

38th Conference on Neural Information Processing Systems (NeurIPS 2024) Track on Datasets and Benchmarks.

 https://doi.org/10.52202/079017-2934

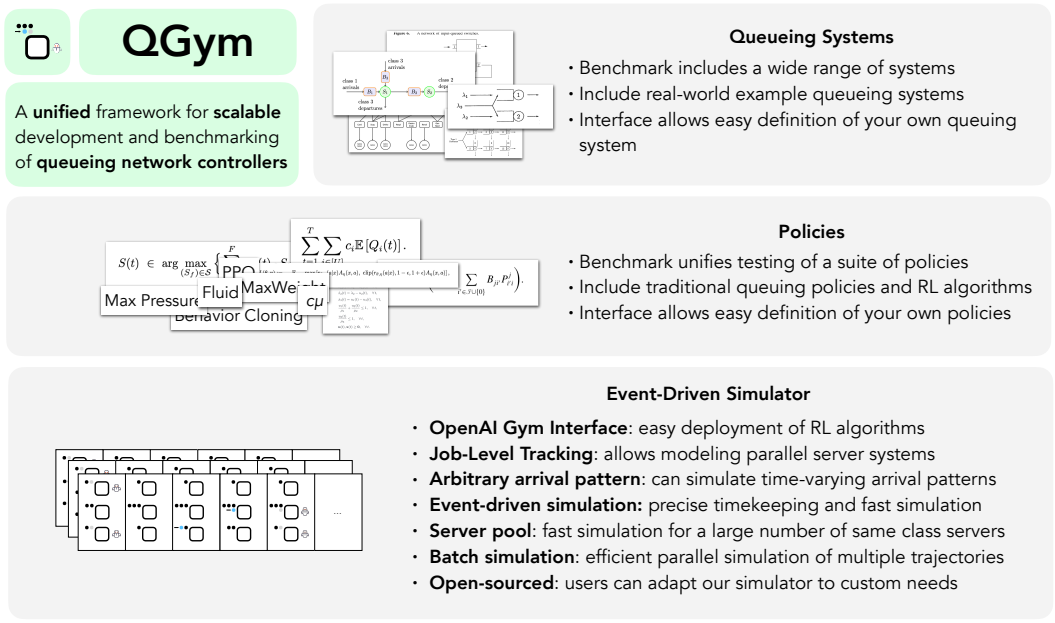

Figure 1: Highlights of QGym framework for developing and benchmarking queuing algorithms. QGym provides an event-driven simulator and benchmarks a wide range of queuing policies and systems. QGym interface also allows users to easily specify new queuing policies and systems.

types, which must be taken into account to make effective scheduling decisions. Second, in systems such as semiconductor fabrication that involve multiple stages of *sequential processing*, congestion can occur at various points in the queuing network, creating bottlenecks that slow down the entire system. Third, workloads are highly non-stationary, featuring predictable and unpredictable demand spikes over time.

There is a large body of work devoted to developing good routing/scheduling policies for queueing networks. The traditional focus of methodological development has been on simple policies with good theoretical performance guarantees in specific network structures. These include (1) load-balancing rules such as joining the shortest queue, and variations such as the power-of-d choices where one randomly samples d queues and sends the job to the shortest one among them [35]; (2) scheduling rules to minimize delay cost such as shortest processing time first, and variations of it such as the $c\mu$-rule [10] and the generalized $c\mu$-rule [30] when we do not know the exact job size and different job classes are associated with different delay costs; and (3) policies that achieve the maximum stability region such as MaxWeight [47] and maximum pressure policies [17].

While easy to implement and interpret, these policies are restrictive to specific queuing architectures or objectives, not data-driven, and difficult to adapt to network-specific features and nonstationarity. As a result, there is a growing interest in using black-box reinforcement learning (RL) techniques to learn queuing controllers in a more data-driven manner, which can better handle realistic settings featuring complex networks with non-stationary workloads.

However, queuing network control poses unique challenges, necessitating new methodological innovation over existing RL algorithms. Compared to typical robotics or game-playing environments, queueing problems have high stochasticity and longer horizons over which the system can become unstable (exploding delays). As a result, model-free RL algorithms have been observed to suffer from instability and substantial stochastic noise in these environments [46, 29, 16].

Another unique challenge is that the queueing system naturally evolves in continuous time, in contrast to the standard discrete time formulation of RL problems. Existing studies assume inter-arrival times and job processing times are exponentially distributed [46, 29, 16]. This so-called Markovian assumption is invoked to represent the queuing network as a discrete-time Markov Decision Process (MDP) with queue lengths as state variables [38]. However, this assumption frequently does not hold in practice, as realistic event times typically exhibit higher variances [40].

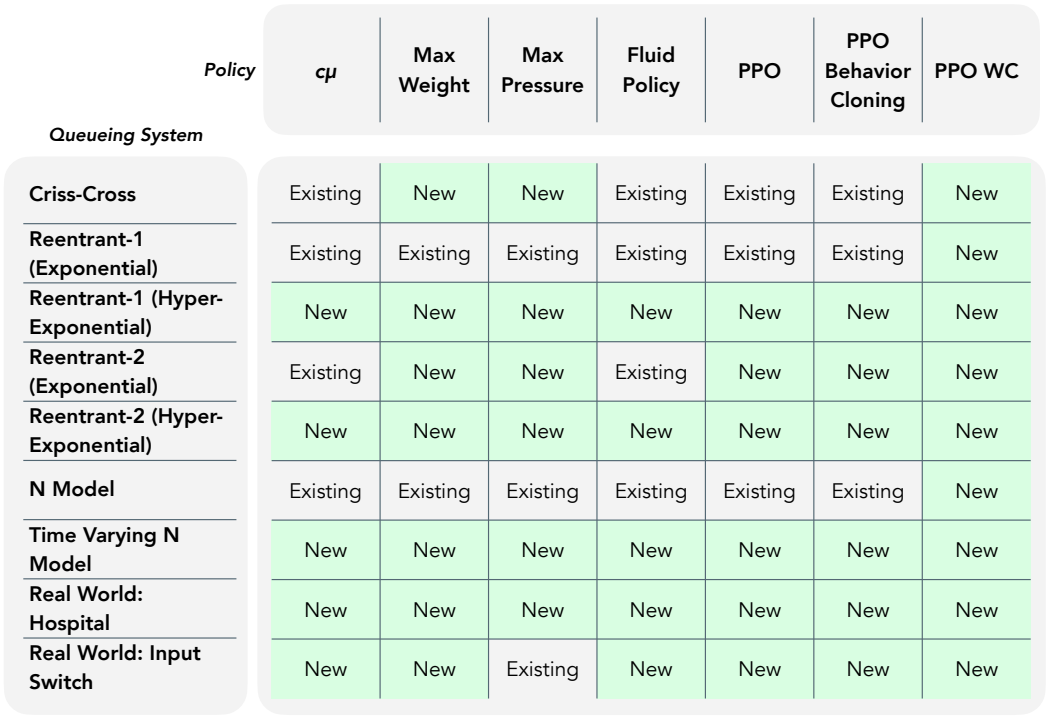

Figure 2: QGym provides a unified and comprehensive benchmarking system for queueing policies, across a range of realistic environments.

Prior works [46, 29, 16] demonstrate the performance of RL algorithms on a small number of problem instances and there is a lack of a common simulation environment or benchmark suite that provides comprehensive evaluations of baselines, including RL algorithms and theory-driven queuing policies. To address this need, we develop a flexible queuing simulation framework (open-sourced and public), QGym, suitable for benchmarking queueing policies across a wide range of problem instances. Our framework can simulate systems with general, non-exponential, and non-stationary event time distributions, requiring only samples from these distributions, which may be obtained from datasets.

This is achieved through the *discrete event dynamical system* representation of queuing networks, a dominant paradigm for queuing simulation (see, e.g., AnyLogic [9], SimPy [32]). Although our queueing setting is markedly different, our framework is broadly inspired by OpenAI Gym [35]. Our modeling framework helps bridge the gap between existing applications of RL, which deal with idealized environments, and industrial simulation paradigms for performance analysis in real-world systems.

We instantiate our abstract and flexible queuing simulation framework with a comprehensive list of queueing environments. Specifically, we consider parallel server systems motivated by skill-based routing problems in service system applications, where the processing speeds depends on both the server type and job type (match of skills) [12]. We also implement the criss-cross network that is widely studied in the queuing control literature [31]. Finally, we consider networks with tandem and reentrant structures that arise in both manufacturing and service systems and is known to suffer from bottleneck resources [28, 26].

The initial set of environments we provide include systems calibrated from real-world applications. For instance, we have a parallel server system with 8 customer classes and 11 different types of servers (server pools) modeling patient flow in the hospital inpatient ward network [20].

Finally, we provide a comprehensive list of baseline policies that span multiple literature. From the classical queueing literature, we implement the $c\mu$-rule, the MaxWeight and maximum pressure policy, and the fluid-based policies [8]. For model-free RL algorithms, we implement several variations of PPO algorithms tailored for queuing network controls. These resources aim to facilitate a thorough and standardized evaluation of RL methods in diverse queuing scenarios.

https://doi.org/10.52202/079017-2934

Taken together, the QGym framework provides the first comprehensive and flexible framework for benchmarking queuing algorithms across a range of different environments. Our initial empirical benchmarking highlight the following considerations that impact the practical performance of RL policies (Sec. 4).

- **Policy architecture is important.** Without any modifications, RL algorithms such as Proximal Policy Optimization (PPO) fail to achieve stability. But equipped with a simple modification inspired by queuing theory, it is able to outperform baseline queuing methods in 77% of instances.
- **Performance gains of RL are larger in noisy, non-exponential environments.** Our modified PPO is able to tailor the policy to the higher noise environment, achieving larger relative gains.
- **Larger networks are still hard to control.** PPO mostly outperforms queuing baselines in small networks, but struggles for larger, more realistic ones

Related Work. Our work is related to three bodies of work. First, it connects to the research developing RL algorithms for queuing network control problems (see, [36, 43, 41, 16, 29, 50] for some recent development). Most of these studies focus on stationary and Markovian systems, and the algorithm performance is empirically tested on a limited set of problem instances. Our work complements this research by creating a flexible queuing simulation framework that can handle more complex systems. QGym offers a diverse set of problems for empirically validating the performance of different RL algorithms and comparing them to standard queueing policies.

	Queuing + RL [36, 43, 41] [16, 29, 50]	Simulators Simio [45] AnyLogic [49]	Ours
Gym environment	✓	✗	✓
Event-driven	✗	✓	✓
General + non-stationary event times	✗	✓	✓
Benchmarking + Policy Implementation	✗	✗	✓

Figure 3: Comparison of our work with related methods

Second, our work is related to research on discrete-event simulation [22, 5, 39] and simulation software such as Simio and AnyLogic. Our work builds on these foundations and extends them to facilitate the benchmarking of RL methods in an *open-sourced, public forum* so that the research community can build on our framework (e.g., crowdsourcing more environments).

Lastly, our work aligns with the growing efforts to build RL library and benchmarking suite for sequential decision-making problems in Operations Research [27, 3, 21]. Our contribution complements these initiatives by focusing specifically on queuing network control problems. This specialization enables us to develop a tailored queuing simulation environment that is highly flexible and capable of addressing a diverse set of queuing control problems. See Table 3 for a comparison between our work and related lines of work.

2 An Event-Driven Queuing Simulation Framework

We implement the following key features in order to design a flexible framework for training and evaluating queueing policies across across diverse environments.

1. Event-Driven Architecture: To address the continuous time nature of the problem, QGym employs an event-driven approach, where system states are updated when new events occur. This enhances the scalability of our framework and allows supporting *arbitrary* arrival patterns, in contrast to traditional discrete time-step-driven models.
2. Extensive Customizability: QGym allows extensive customization in the queuing network topology, job processing pipelines, and stochastic inputs, enabling a broad set of queuing systems that meet the needs of both academic and industrial applications (see Sec. 4).
3. OpenAI Gym Integration: Built on the OpenAI Gym interface, QGym facilitates easy testing and deployment of diverse queuing policies, both RL-based and traditional. Its modular design promotes easy integration of new functionalities, supporting continuous evolution.

Event-Driven MDP Formulation We begin by describing how to convert a classical multi-class queuing scheduling problem into an *event-driven* MDP problem. Consider a queuing system with M

queues and N servers. Assume a Poisson arrival for each queue with an arrival rate λ_i for $i \in [M]$. Jobs within a queue are processed on a first-come-first-served basis. A decision-maker can assign a job from queue i to server j if j is available, at a service rate μ_{ij}. Each server can serve only one job at a time and each queue can only be served by one server at a time. Each job in queue i incurs a holding cost c_i per unit of time. The objective is to design rules to match jobs with servers that minimize the total holding cost over a finite time horizon.

For the MDP formulation, consider step k associated with timestamp t_k. Let $Q_i(t_k) \in \mathbb{N}$, for $i \in [M]$, represent the queue length of queue i at step k. Let $\tau_i^{\mathrm{A}}(t_k) \in \mathbb{R}$, for $i \in [M]$, denote the *residual inter-arrival time* at queue i from time t_k. Define the action $a(t_k) \in \{0,1\}^{N\times M}$ to decide the assignment jobs to servers at step k where $a_{ij}(t_k) = 1$ if a job at queue i is assigned to server j; otherwise, it is 0, subject to feasibility. After making the decision, $\tau_i^{\mathrm{S}}(t_k)/\mu_{ij} \in \mathbb{R}$ is the *residual service time* for queue i if server j is assigned (the time is infinite if the job is not assigned to a server). To express this compactly, we define $\mu_i(t_k) \equiv \mu_i^\top a_i(t_k)$, with μ_i and $a_i(t_k)$ being the ith rows, and refer to the service time as $\tau_i^{\mathrm{S}}(t_k)/\mu_i(t_k)$, which will be equal to ∞ if no server is assigned to queue i. The state of the MDP is the tuple $s_k = (t_k, Q(t_k), \tau^A(t_k), \tau^S(t_k))$.

It is worth emphasizing that almost always the controller only observes the queue-length $Q(t_k)$. Considering that our framework involves an expanded state space that includes residual event times, our framework is technically a *partially observable Markov Decision Process* (POMDP), with observations o_k only consisting of the current time-stamp t_k and the queue-lengths $Q(t_k)$,

$$o_k = (t_k, Q(t_k)), \qquad s_k = (t_k, Q(t_k), \tau^A(t_k), \tau^S(t_k)).$$

The next event is the event with the minimum remaining processing time, which we denote as the inter-epoch time τ_{k+1},

$$\tau_{k+1} = \min\left\{\tau_i^{\mathrm{A}}(t_k), \frac{\tau_i^S(t_k)}{\mu_i(t_k)} | i \in [M]\right\}$$

The event determines the update to the state. If an arrival to queue i occurs, then the queue-length $Q(t_{k+1})$ is incremented by 1 in the ith position which is represented by adding the canonical basis vector e_i. If a service occurs, then the queue-length $Q(t_{k+1})$ will be updated by the vector $\Delta_i \in \{1, 0, -1\}^N$. Typically, this update will involve decrementing the queue-length in the ith position by 1 since a job departs from queue i. At the same time, this could increment the queue-length of another queue, as is the case for tandem queues where jobs are routed to other queues after processing.

$$Q(t_{k+1}) = Q(t_k) + \sum_{i=1}^{N} e_i \mathbf{1}\{\tau_i^A(t_k) = \tau_{k+1}\} + \sum_{i=1}^{N} \Delta_i \mathbf{1}\left\{\frac{\tau_i^S(t_k)}{\mu_i(t_k)} = \tau_{k+1}\right\}. \tag{1}$$

After the event occurs, the residual event times are reduced by τ_{k+1}, which is the time that elapsed between events. For the event which occurred, the residual event time are reset by drawing new event times from a distribution. We denote the new event times as T_i^{A} if the event was an arrival and T_i^{S} if the event was a service,

$$\tau_i^{\mathrm{A}}(t_{k+1}) = \tau_i^{\mathrm{A}}(t_k) - \tau_{k+1} + T_i^A \cdot \mathbf{1}\{\tau_i^A = \tau_{k+1}\} \tag{2}$$

$$\tau_i^{\mathrm{S}}(t_{k+1}) = \tau_i^{\mathrm{S}}(t_k) - \mu_i(t_k)\tau_{k+1} + T_i^S \cdot \mathbf{1}\left\{\frac{\tau_i^S(t_k)}{\mu_i(t_k)} = \tau_{k+1}\right\} + \infty \cdot \mathbf{1}\{Q(t_{k+1}) = 0\}. \tag{3}$$

Since services cannot occur to an empty queue, the service time is infinite if the queue-length is zero.

This event-driven MDP formulation advances time steps by events, i.e., the next event time, in contrast to existing RL work in queuing that often uses a time-step-driven MDP which poses scalability challenges in real-world applications [34].

Functionality We implement functionalities to generalize the classical multi-class, multi-server problem described above. This enables QGym to accommodate an extensive range of queuing problems in real-world applications (e.g., see the hospital example in Sec. 4). While not exhaustive, our system is designed to be flexible, allowing new models to be easily incorporated and tested.

 https://doi.org/10.52202/079017-2934

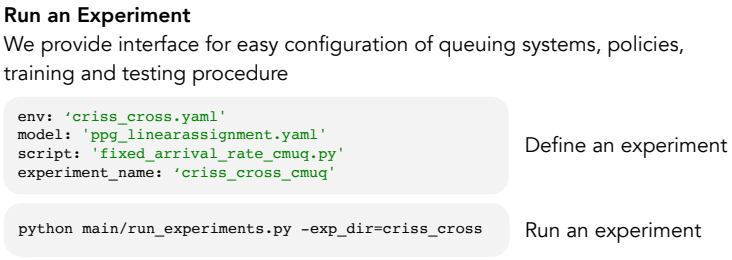

Figure 4: QGym provides an user-friendly interface to define and run experiments for evaluating routing policies on queuing networks.

Network Topology. We allow customized network topologies. Given a binary matrix $B \in \mathbb{R}^{M \times N}$, server j is permitted to serve jobs from queue i only when $B_{ij} = 1$.

Job Transition. Completed jobs from queue $i \in [M]$ can transition to another queue $i' \in [M]$ instead of leaving the system. This facilitates complex job processing pipelines, such as re-entrant and tandem queues.

Arbitrary Arrivals. Users can define arbitrary arrival patterns for queues via a Python function that inputs the current time and outputs the time until the next arrival. This feature allows for the simulation of time-varying and non-Poisson arrivals. Arrivals "generated" from real data are also supported.

Service Time Distribution. Service times are drawn from arbitrary distributions specified by the user, with service rates as parameters. Although time-varying service rates are not yet supported, they can be implemented in a manner similar to arrival patterns.

Server Pool. Users can define each class of server as a pool. When many servers share the same characteristics, users can specify the number of servers in each class (server pool) instead of creating numerous separate servers and inflating the size of the network matrix. This mechanism enables the simulation of large-scale systems without compromising performance.

Job-Level Tracking. The simulator tracks states at the job level, monitoring the service time for each job in a queue. The fine-grained job-level tracking enables simulation of parallel-server systems, where multiple servers can serve a single queue. To illustrate why a less fine-grained choice of tracking at queue level could fail, we consider the cases where multiple servers serve a single queue and the jobs are preempted. Only tracking at queue level does not allow recording remaining service time for each job and resuming service them later. Tracking states at job level enables flexible simulations of parallel server systems.

Reward. Users can define rewards using arbitrary functions on states and actions. In this paper, we focus on minimizing holding costs as a representative example.

OpenAI Gym Design. The simulator environment is structured as an OpenAI Gym environment, adhering to its design principles. This allows users to train and test a variety of reinforcement learning algorithms seamlessly. Each simulation trajectory consists of a sequence of steps (defined in the OpenAI Gym `step` format) and supports batch-based GPU execution to accelerate computation. We provide a range of environments (e.g., N-model, reentrant, re-reentrant, criss-cross, etc.) and policies, including both RL methods and traditional ones (see Sec. 3), to facilitate easy testing. Users can conveniently configure environment and policy parameters using `.yaml` files, and new environments and policies can be added by following the OpenAI Gym convention. See Figure 4 for code snippets of our user-friendly interface for defining and runnning experiments. More details can be found in the Appendix C.

3 Benchmark Policies

In this section, we introduce the queuing policies benchmarked in our testbed. Formally, each policy $\pi(\cdot|o)$ maps an observation $o \in \mathbb{N}^N$ (the current queue-lengths) to an action $a \in \{0,1\}^{N \times M}$. These include both traditional control-based policies and RL-based policies tailored for queuing.

3.1 Traditional Policies

All traditional policies considered fall into the class of policies using the linear assignment rule: given a policy π, a priority matrix $\rho \in \mathbb{R}^{M\times N}$ is outputted from π at each step, the action (i.e., job-server assignment) is then decided by

$$\max_{a\in\mathcal{A}} \sum_{i,j} \rho_{ij} a_{ij}$$

where $\mathcal{A} \subset \mathbb{R}^{M\times N}$ captures the feasibility (e.g., compatibility and resource capacity) constraints.

$c\mu$-rule. A classic policy for scheduling multiple classes of jobs is the $c\mu$-rule, which has been shown to minimize the linear waiting cost in multi-class single-server queues [15]. In this case, $\rho_{ij} = c_i\mu_{ij}$. Server j prioritizes the queue with a larger $c_i\mu_{ij}$-index, where c_i denotes the holding cost per job per unit time for queue i, and μ_{ij} is the service rate when server j processes a job from queue i.

MaxWeight. Another important class of policies is known as MaxWeight policies, which has been shown to be maximally stable for single-hop networks [48] and are also known for their favorable asymptotic properties under a resource pooling condition [47]. We consider a specific form of MaxWeight policy where $\rho_{ij} = c_iQ_i\mu_{ij}$ with Q_i being the queue length of queue i. Here, by taking the queue lengths into account, we are able to better balance the workload in the system.

Maximum pressure. The maximum pressure policies, which are also known as the back pressure policies, are similar to the MaxWeight policies but account for workload externality within the network. This additional consideration allows for better workload balancing in the multihop setting, especially in networks with tandem or reentrant structures. These policies have been shown to be maximally stable in multi-hop networks [48, 17]. We consider a specific form of the maximum pressure policy under which $\rho_{ij} = (c_iQ_i\mu_{ij} - \sum_{k=1}^{M} c_kQ_k\mu_{ij}p_{ik})$, where p_{ik} is the probability that after a class i job is processed by server j, it will join queue k next. Note that when $p_{ik} = 0$ for all k, the maximum pressure policy simplifies to the MaxWeight policy. However, when $p_{ik} > 0$ for some k, the maximum pressure policy accounts for the fact that processing a class i job will generate a class k job, thus considering the impact on "downstream" queues.

Fluid Policy. One can derive a 'fluid model' of the queuing network as a system of ordinary differential equations (ODEs) driven by the service and arrival rates of the network [11]. By discretizing the ODEs on a finite grid, one can minimize the linear holding costs by solving a linear program (LP). We then use the computed priorities ρ_{ij} in the original queuing network. To maintain fidelity with the original dynamics, we periodically re-solve the LP. We solve the LP via CVX [19], and resolve after every 1000 steps.

3.2 Deep RL based methods

Proximal Policy Optimization (PPO) has been a popular choice for applying RL to queuing [29, 16]. Following the convention, we implemented a few variants of PPO in our testbed. We apply existing and develop new modifications to improve the stability and scalability of PPO in queuing systems.

PPO. The action space in our problem is $a \in \mathbb{R}^{M\times N}$. Thus, directly applying vanilla PPO [42] will suffer from the explosion of dimensionality. To address this issue, we require $\{a_{ij}\}_{i=1}^{M}$ to be a probability distribution for all $j \in [N]$. We sample $a'_j \sim \{a_{ij}\}_{i=1}^{M}$ for each j independently to decide which queue server j serves. The feasibility constraint is then verified by the environment. Compared to the existing queuing RL method that discretizes the action space [16], this parameterization is much more scalable when M and N both grow. The classic normalization tricks have been implemented to make PPO more stable (e.g., advantage function normalization, reward normalization, etc.). In addition, to reduce the variance for the Generalized Advantage Estimation (GAE) in PPO

$$\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} := \sum_{l=0}^{T} (\gamma\lambda)^l \delta_{t+l}^V \text{ where } \delta_t^V = r_t + \gamma V(s_{t+1}) - V(s_t),$$

we truncate T to T_0 where T_0 is the first time that all queues are empty (i.e., regenerative point).

PPO with Behavior Cloning (PPO-BC). Implementing the PPO described above with a random initial policy still suffers from poor performance due to instability. Similar to [16], we address this by using behavior cloning. We first train π to imitate a Max-Weight style policy that assigns servers

https://doi.org/10.52202/079017-2934

to classes with probability proportional to e^{Q_i}. Using this procedure as a warm-up significantly enhances the stability of training and achieves much better results compared to PPO alone.

Work-Conserving PPO (PPO-WC). We have a simple observation: the policy should never assign server capacity to an empty queue (so-called 'work conservation' rule [18]). We impose this 'inductive bias' to the policy design directly. To do so, we mask the probabilities a_{ij} obtained from PPO while preserving differentiability:

$$a_{ij} = \frac{a_{ij}\mathbf{1}\{Q_i > 0\}}{\sum_{i=1}^{M} a_{ij}\mathbf{1}\{Q_i > 0\}}$$

where Q_i is the length of queue i when taking actions. In case the queues are all empty, we avoid division-by-zero errors by clipping the denominator for some small ϵ. As we observe in the experiments in Section 4, this small change greatly improves the performance of PPO. With randomly initialized policy parameters, training algorithms utilizing WC policy parameterization consistently outperform those using vanilla parameterization. Notably, the PPO-WC training algorithm demonstrates a high training stability, such that action clipping is almost never required, which was the core advantage of PPO. To further validate the effectiveness and advantages of WC parameterization, we implemented **A2C** (a vanilla actor-critic algorithm without clipping and KL regularization) with the same WC parameterization and observed comparable performance to that of PPO-WC. These results underscore the robustness and generalizability of WC parameterization.

4 Experiments

Using our environment, we benchmark the performance of PPO and traditional queuing baselines across a diverse suite of queuing networks. We curate a set of queuing network instances, drawing upon networks studied in the queuing literature as well as novel instances, with coverage of network architectures relevant to manufacturing, hospital patient flow, and wireless network applications. Overall, we observe that while PPO alone performs quite poorly, *PPO-WC outperforms the traditional policies in 77% of all instances*, highlighting the importance of incorporating queuing structure in RL policy design.

4.1 Setup

Network structure. In total, we consider 20 unique problem instances across the following networks. See Fig. 5 for the corresponding network topologies.

(a) **Hospital**: Patients arrive to $M = 8$ specialties (Cardiology, Surgery, Orthopedics, Respiratory disease, Gastroenterology and endoscopy, Renal disease, General Medicine, Neurology) split across 11 inpatient wards. Each ward consists of multiple beds (servers). In total, this is modeled by $N = 497$ servers across the 11 wards. The hospital employs a focused-care model where each ward is primarily designated to serve patients from one specialty or two specialties. The network topology, arrival rates, and service rates are calibrated to a real hospital setting.

(b) **Input-Queue Switch** [17, 33]: Packets in a crossbar switch arrive to $M = 6$ queues, and are processed by $N = 3$ servers.

(c) **Reentrant (L)** [8, 16]: Manufacturing lines process goods in several sequential steps L. We consider a family of instances with $L \in \{2, ..., 10\}$ with $M = 3L$ queues and $N = L$ servers. High variance in the service times can lead to bottlenecks in the network, and so we also consider instances with hyper-exponential service times, which are mixtures of exponential distributions.

(d) **Five-by-Five Network** [13]: Call-centers route customers from $M = 5$ classes to $N = 5$ servers. Call center demand changes throughout the day, which we model through time-varying inter-arrival times $\tau^A \sim \text{Exp}(\lambda(t))$.

(e) **Criss-Cross** [25, 6, 31]: A standard reentrant network considered in the literature, consisting of $M = 3$ queues and $N = 2$ servers.

(f) **N-model** [24]: A standard parallel-server system considered in the literature, consisting of $M = 2$ queues and $N = 2$ servers.

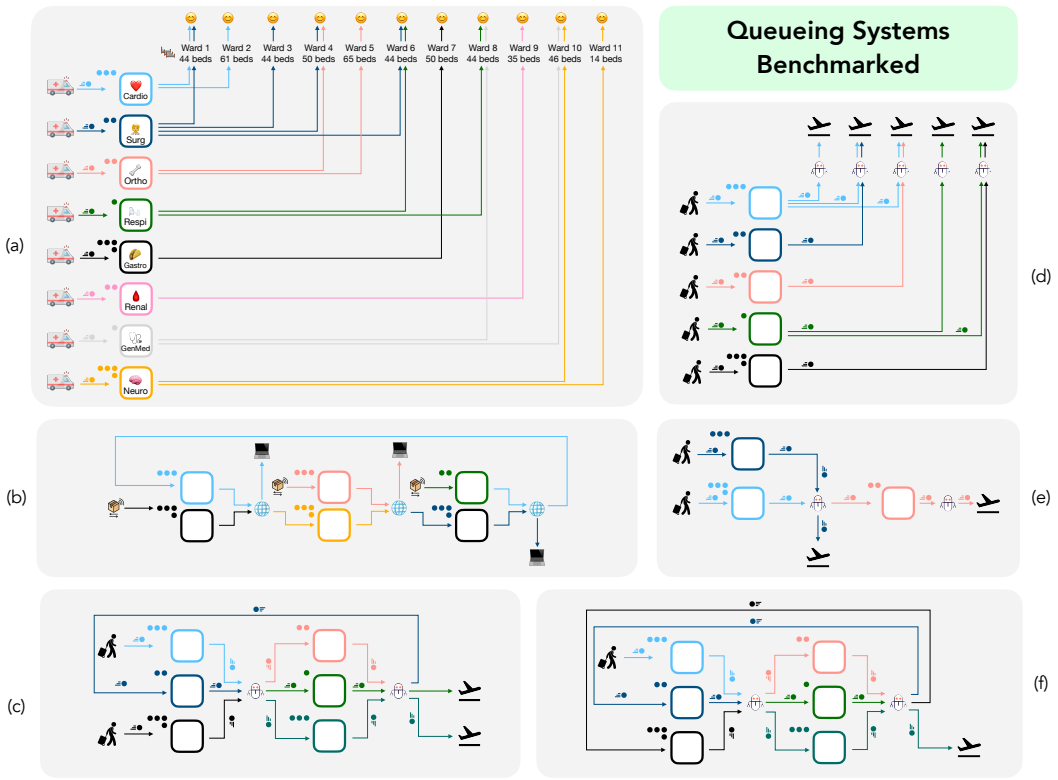

Figure 5: Queuing systems QGym benchmarks. (a) Real-world example of hospital routing. (b) Real-world example of data input-switch routing. (c) and (f) Two variations of reentrant networks. (d) Five-by-Five netowrk for modeling call centers. (e) Criss-cross network. See details in section 4

Objective. The core performance metric we consider is the long-run-average total queue-length, which is approximated by averaging over a long horizon of n events.

$$\mathbb{E}\left[\frac{1}{t_n}\sum_{k=1}^{n}\sum_{i=1}^{N} Q_i(t_k)(t_{k+1}-t_k)\right] = \mathbb{E}\left[\frac{1}{t_n}\int_0^{t_n}\sum_{i=1}^{N} Q_i(t)dt\right] \tag{4}$$

For each policy, we estimate the expected time-average total queue length by evaluating the policy over 100 trajectories. We also report the corresponding standard errors.

Training Procedure All PPO variants were trained under the same conditions. Each policy was trained over 100 episodes, each consisting of $50,000$ environment steps parallelized over 50 actors. Following existing works [16], we used a discount factor of $\gamma = 0.998$, a GAE parameter of $\lambda = 0.99$, and set the KL divergence penalty of $\beta = 0.03$. For the value network, we used a batch size of 2500, while for the policy network, we used the entire rollout buffer (batch size of $50,000$) to take one gradient step. We performed 3 PPO gradient updates on the same rollout data. For all the experiments, we used Adam optimizer with a cosine decaying warming-up learning rate scheduler. The learning rates were set to 3×10^{-4} for the value network and 9×10^{-4} for the policy network. We used 3% of the training horizon to warm up to the maximum learning rate and then cosine decayed to 1×10^{-5} for both networks.

4.2 Results

Tables 1-5 document the time-averaged total queue length for the policies we consider. Our systematic benchmarking illustrates the differences in practical performance of reinforcement learning and traditional queuing policies. Our findings can be summarized into three folds.

PPO learns an effective controller, but only under the right policy architecture. Without any modifications, in every setting PPO fails to stabilize the queuing network, systematically confirming an observation made in previous works [29]. Behavior cloning a stabilizing policy drastically improves the training process, yet the policy still fails to achieve parity with the traditional queuing methods. It is only when we endow PPO with a work-conserving policy, that we are able to completely stabilize the training process and surpass the performance of traditional queuing methods in most

https://doi.org/10.52202/079017-2934

Table 1: Criss Cross

Network	$c\mu$	MW	MP	FP	PPO	PPO BC	PPO WC	A2C WC
Criss Cross BH	16.1 ± 0.3	$\mathbf{15.3 \pm 0.3}$	19.0 ± 0.3	18.2 ± 2.7	8.6E+3 $\pm$ 4.6	24.0 ± 0.2	$\mathbf{15.4 \pm 0.2}$	15.29 ± 0.2

Table 2: Reentrant [Exponential]

L	$c\mu$	MW	MP	FP	PPO	PPO BC	PPO WC	A2C WC
2	19.0 ± 0.4	14.8 ± 0.4	18.9 ± 0.5	16.8 ± 4.3	1.8E+3 $\pm$ 6.6	25.1 ± 0.6	13.6 ± 0.4	$\mathbf{13.01 \pm 0.3}$
3	$\mathbf{21.6 \pm 0.6}$	24.8 ± 0.7	30.9 ± 1.0	27.7 ± 4.4	1.0E+4 $\pm$ 21.6	48.2 ± 0.5	22.6 ± 0.4	22.0 ± 0.3
4	30.1 ± 0.9	32.1 ± 1.1	40.3 ± 1.4	40.6 ± 4.8	1.8E+4 $\pm$ 41.7	183.4 ± 5.2	$\mathbf{29.7 \pm 0.4}$	30.7 ± 0.4
5	51.3 ± 1.3	50.0 ± 1.5	52.2 ± 1.6	49.8 ± 5.1	2.7E+4 $\pm$ 84.4	135.2 ± 3.1	$\mathbf{38.7 \pm 0.4}$	39.1 ± 0.5
6	54.7 ± 1.6	49.2 ± 1.3	59.1 ± 2.1	54.5 ± 4.2	5.9E+4 $\pm$ 315.3	358.0 ± 9.7	48.5 ± 0.5	$\mathbf{47.4 \pm 0.5}$
7	56.4 ± 1.6	$\mathbf{54.4 \pm 1.8}$	70.5 ± 2.9	63.7 ± 6.7	4.4E+4 $\pm$ 208.1	526.6 ± 8.7	56.3 ± 8.2	56.7 ± 0.8
8	$\mathbf{59.4 \pm 2.2}$	68.0 ± 1.7	81.4 ± 3.0	74.0 ± 6.7	5.9E+4 $\pm$ 315.2	868.5 ± 6.0	65.8 ± 6.2	67.5 ± 0.6
9	72.7 ± 2.5	$\mathbf{64.4 \pm 2.1}$	90.8 ± 3.2	83.1 ± 7.1	1.1E+5 $\pm$ 2219.7	1304.5 ± 10.1	75.8 ± 0.7	77.0 ± 0.8
10	87.7 ± 2.7	$\mathbf{80.1 \pm 1.9}$	100.5 ± 3.2	93.6 ± 8.0	1.6E+5 $\pm$ 852.8	3809.1 ± 10.4	83.1 ± 0.7	90.2 ± 0.9

Table 3: Reentrant [Hyperexponential]

L	$c\mu$	MW	MP	FP	PPO	PPO BC	PPO WC	A2C WC
2	31.69 ± 1.3	$\mathbf{22.40 \pm 1.2}$	43.8 ± 1.8	43.6 ± 7.5	9.9E+3 $\pm$ 20.7	62.7 ± 1.3	29.9 ± 0.7	30.2 ± 0.7
3	$\mathbf{36.76 \pm 1.9}$	43.00 ± 2.2	68.7 ± 2.7	59.2 ± 8.2	19.6E+3 $\pm$ 58.0	305.1 ± 13.8	47.5 ± 0.8	47.8 ± 1.1
4	$\mathbf{58.58 \pm 2.5}$	74.54 ± 2.8	89.4 ± 3.6	75.6 ± 15.3	18.9E+3 $\pm$ 53.1	167.2 ± 5.1	64.4 ± 1.2	62.8 ± 1.4
5	$\mathbf{68.91 \pm 4.0}$	73.19 ± 3.7	112.0 ± 4.9	97.0 ± 12.9	48.0E+3 $\pm$ 153.5	913.4 ± 19.9	81.8 ± 1.1	84.9 ± 1.2
6	$\mathbf{85.16 \pm 4.7}$	98.75 ± 3.9	126.7 ± 6.2	111.2 ± 14.4	59.1E+3 $\pm$ 336.4	2383.0 ± 15.2	99.8 ± 1.5	100.8 ± 1.4
7	$\mathbf{100.24 \pm 5.9}$	119.01 ± 3.9	152.3 ± 6.6	151.0 ± 21.3	65.4E+3 $\pm$ 325.9	3054.6 ± 16.6	118.2 ± 2.0	120.5 ± 2.1

Table 4: Parallel Server

Network	$c\mu$	MW	MP	FP	PPO	PPO BC	PPO WC	A2C WC
N Model	1.7E+2 $\pm$ 12.3	$\mathbf{40.2 \pm 2.2}$	$\mathbf{40.2 \pm 2.2}$	7.9E+2 $\pm$ 18.8	8.8E+3 $\pm$ 28.5	100.9 ± 1.9	44.3 ± 1.8	49.8 ± 0.2
Five-by-Five	17.8 ± 1.2	$\mathbf{15.2 \pm 0.7}$	$\mathbf{15.2 \pm 0.7}$	26.7 ± 2.5	1.2E+4 $\pm$ 15.3	25.2 ± 0.1	16.8 ± 0.2	162.88 ± 4.8

Table 5: Real World Example

Network	$c\mu$	MW	MP	FP	PPO	PPO BC	PPO WC	A2C WC
Input Switch	$\mathbf{5.3 \pm 0.0}$	5.6 ± 0.0	4.9E+3 $\pm$ 18.9	4.9E+3 $\pm$ 12.2	7.3E+3 $\pm$ 8.9	11.8 $\pm$ 0.1	$\mathbf{5.3 \pm 0.3}$	5.4 ± 0.0
Hospital	$\mathbf{4.4E+2 \pm 0.0}$	$\mathbf{4.4E+2 \pm 8.9}$	$\mathbf{4.4E+2 \pm 8.9}$	1.5E+3 $\pm$ 4.9	2.4E+4 $\pm$ 6.6	2.5E+4 $\pm$ 7.0	2.3E+4 $\pm$ 7.4	2.3E+4 $\pm$ 7.3

settings (the pair-wise win rate is 77%). This suggests that the usual 'tabula rasa' [1] approach of reinforcement learning, which aims to learn from a completely flexible and unstructured policy class, is unsuitable for queuing control. The 'inductive bias' of work-conservation not only speeds up learning, but is decisive in enabling the algorithm to improve upon existing policies.

Performance gains from PPO are larger in noisier, non-exponential environments. We compare the results in Table 2, which details performance for the reentrant networks under exponentially-distributed event times with Table 3, which involves hyper-exponential noise. In Table 2, the improvements of PPO-WC over other baselines are relatively modest, around a 10% improvement at most. Yet, Table 3 shows that under hyper-exponential noise, the improvements can be larger, resulting in a relative reduction in holding costs of around 21% with $L = 2$. We are only able to observe these improvements because our simulation framework can incorporate general noise inputs.

There is plenty of room for improvement. PPO-WC mainly outperforms queuing baselines on small-scale examples, but incurs a higher cost in more realistic, larger-scale instances such as the hospital network, despite being trained over 5 million steps. This gap in performance points towards the sample-inefficiency of RL in larger networks.

Additional experiment results and a detailed setup for reproducibility is provided in Appendix B, A, and D.1.

5 Conclusion

We propose a simulation framework and a comprehensive suite of benchmarks for queuing network control. Although RL is capable of training effective controllers, there still remain performance gaps compared to standard baselines and we hope that our simulation framework can enable algorithmic progress towards bridging these gaps.

References

[1] Agarwal, R., Schwarzer, M., Castro, P. S., Courville, A. C., and Bellemare, M. (2022). Reincarnating reinforcement learning: Reusing prior computation to accelerate progress. *Advances in Neural Information Processing Systems*, 35:28955–28971.

[2] Aksin, Z., Armony, M., and Mehrotra, V. (2007). The modern call center: A multi-disciplinary perspective on operations management research. *Production and operations management*, 16(6):665–688.

[3] Archer, C., Banerjee, S., Cortez, M., Rucker, C., Sinclair, S. R., Solberg, M., Xie, Q., and Lee Yu, C. (2022). Orsuite: Benchmarking suite for sequential operations models. *ACM SIGMETRICS Performance Evaluation Review*, 49(2):57–61.

[4] Armony, M., Israelit, S., Mandelbaum, A., Marmor, Y. N., Tseytlin, Y., and Yom-Tov, G. B. (2015). On patient flow in hospitals: A data-based queueing-science perspective. *Stochastic systems*, 5(1):146–194.

[5] Asmussen, S. and Glynn, P. W. (2007). *Stochastic simulation: algorithms and analysis*, volume 57. Springer.

[6] Avram, F., Bertsimas, D., and Ricard, M. (1995). Fluid models of sequencing problems in open queueing networks; an optimal control approach. *Institute for Mathematics and its Applications*, 71:199.

[7] Banerjee, S., Freund, D., and Lykouris, T. (2022). Pricing and optimization in shared vehicle systems: An approximation framework. *Operations Research*, 70(3):1783–1805.

[8] Bertsimas, D., Nasrabadi, E., and Paschalidis, I. C. (2014). Robust fluid processing networks. *IEEE Transactions on Automatic Control*, 60(3):715–728.

[9] Borshchev, A. (2014). Multi-method modelling: Anylogic. *Discrete-event simulation and system dynamics for management decision making*, pages 248–279.

[10] Buyukkoc, C., Varaiya, P., and Walrand, J. (1985). The cμ rule revisited. *Advances in applied probability*, 17(1):237–238.

[11] Chen, H. and Yao, D. D. (1993). Dynamic scheduling of a multiclass fluid network. *Operations Research*, 41(6):1104–1115.

[12] Chen, J., Dong, J., and Shi, P. (2020). A survey on skill-based routing with applications to service operations management. *Queueing Systems*, 96:53–82.

[13] Chen, J., Dong, J., and Shi, P. (2023). Optimal routing under demand surges: The value of future arrival rates. *Operations Research*.

[14] Cont, R., Stoikov, S., and Talreja, R. (2010). A stochastic model for order book dynamics. *Operations research*, 58(3):549–563.

[15] Cox, D. and Smith, W. (1961). *Queues*. Methuen, London, 5 edition.

[16] Dai, J. G. and Gluzman, M. (2022). Queueing network controls via deep reinforcement learning. *Stochastic Systems*, 12(1):30–67.

[17] Dai, J. G. and Lin, W. (2005). Maximum pressure policies in stochastic processing networks. *Operations Research*, 53(2):197–218.

[18] Dai, J. G. and Meyn, S. P. (1995). Stability and convergence of moments for multiclass queueing networks via fluid limit models. *IEEE Transactions on Automatic Control*, 40(11):1889–1904.

[19] Diamond, S. and Boyd, S. (2016). Cvxpy: A python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83):1–5.

[20] Dong, J., Shi, P., Zheng, F., and Jin, X. (2020). Structural estimation of load balancing behavior in inpatient ward network. Technical report, Working paper.

 https://doi.org/10.52202/079017-2934

[21] Eckman, D. J., Henderson, S. G., and Shashaani, S. (2023). Simopt: A testbed for simulation-optimization experiments. *INFORMS Journal on Computing*, 35(2):495–508.

[22] Fishman, G. S. (2001). *Discrete-event simulation: modeling, programming, and analysis*, volume 537. Springer.

[23] Harchol-Balter, M. (2013). *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press.

[24] Harrison, J. M. (1998). Heavy traffic analysis of a system with parallel servers: asymptotic optimality of discrete-review policies. *The Annals of Applied Probability*, 8(3):822–848.

[25] Harrison, J. M. and Wein, L. M. (1990). Scheduling networks of queues: Heavy traffic analysis of a two-station closed network. *Operations research*, 38(6):1052–1064.

[26] Huang, J., Carmeli, B., and Mandelbaum, A. (2015). Control of patient flow in emergency departments, or multiclass queues with deadlines and feedback. *Operations Research*, 63(4):892–908.

[27] Hubbs, C. D., Perez, H. D., Sarwar, O., Sahinidis, N. V., Grossmann, I. E., and Wassick, J. M. (2020). Or-gym: A reinforcement learning library for operations research problems. *arXiv preprint arXiv:2008.06319*.

[28] Li, J. and Meerkov, S. M. (2008). *Production systems engineering*. Springer Science & Business Media.

[29] Liu, B., Xie, Q., and Modiano, E. (2022). Rl-qn: A reinforcement learning framework for optimal control of queueing systems. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 7(1):1–35.

[30] Mandelbaum, A. and Stolyar, A. L. (2004). Scheduling flexible servers with convex delay costs: Heavy-traffic optimality of the generalized cμ-rule. *Operations Research*, 52(6):836–855.

[31] Martins, L. F., Shreve, S. E., and Soner, H. M. (1996). Heavy traffic convergence of a controlled, multiclass queueing system. *SIAM journal on control and optimization*, 34(6):2133–2171.

[32] Matloff, N. (2008). Introduction to discrete-event simulation and the simpy language. *Davis, CA. Dept of Computer Science. University of California at Davis. Retrieved on August*, 2(2009):1–33.

[33] McKeown, N. (1999). The islip scheduling algorithm for input-queued switches. *IEEE/ACM transactions on networking*, 7(2):188–201.

[34] Menda, K., Chen, Y.-C., Grana, J., Bono, J. W., Tracey, B. D., Kochenderfer, M. J., and Wolpert, D. (2018). Deep reinforcement learning for event-driven multi-agent decision processes. *IEEE Transactions on Intelligent Transportation Systems*, 20(4):1259–1268.

[35] Mitzenmacher, M. (2001). The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104.

[36] Moallemi, C. C., Kumar, S., and Van Roy, B. (2008). Approximate and data-driven dynamic programming for queueing networks. *working paper*.

[37] Neely, M. (2022). *Stochastic network optimization with application to communication and queueing systems*. Springer Nature.

[38] Neely, M. J. (2010). *Stochastic Network Optimization with Application to Communication and Queueing Systems*. Number 7 in Synthesis Lectures on Communication Networks. Morgan and Claypool Publishers.

[39] Nelson, B. L. (2010). *Stochastic modeling: analysis & simulation*. Courier Corporation.

[40] Paxson, V. and Floyd, S. (1995). Wide area traffic: the failure of poisson modeling. *IEEE/ACM Transactions on networking*, 3(3):226–244.

[41] Qu, G., Wierman, A., and Li, N. (2020). Scalable reinforcement learning of localized policies for multi-agent networked systems. In *Learning for Dynamics and Control*, pages 256–266. PMLR.

[42] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

[43] Shah, D., Xie, Q., and Xu, Z. (2020). Stable reinforcement learning with unbounded state space. *arXiv preprint arXiv:2006.04353*.

[44] Shanthikumar, J. G., Ding, S., and Zhang, M. T. (2007). Queueing theory for semiconductor manufacturing systems: A survey and open problems. *IEEE Transactions on Automation Science and Engineering*, 4(4):513–522.

[45] Simio LLC (2024). Simio simulation software. `https://www.simio.com`. Accessed: 2024-06-05.

[46] Singh, S. and Bertsekas, D. (1996). Reinforcement learning for dynamic channel allocation in cellular telephone systems. *Advances in neural information processing systems*, 9.

[47] Stolyar, A. L. (2004). Maxweight scheduling in a generalized switch: State space collapse and workload minimization in heavy traffic. *The Annals of Applied Probability*, 14(1):1–53.

[48] Tassiulas, L. and Ephremides, A. (1990). Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks. In *29th IEEE Conference on Decision and Control*, pages 2130–2132. IEEE.

[49] The AnyLogic Company (2024). Anylogic simulation software. `https://www.anylogic.com`. Accessed: 2024-06-05.

[50] Wei, H., Liu, X., Wang, W., and Ying, L. (2024). Sample efficient reinforcement learning in mixed systems through augmented samples and its applications to queueing networks. *Advances in Neural Information Processing Systems*, 36.

 https://doi.org/10.52202/079017-2934

A Code release

Our code is available at: `https://github.com/namkoong-lab/QGym`

B Additional Benchmark Result

We provide the benchmark results for reentrant-2 [exponential] and reentrant-2 [hyperexpenential] queuing systems in Table 6 and 7.

Table 6: Reentrant-2[Exp]

Network	$c\mu$	MW	MP	FP	PPO	PPO BC	PPO WC
2	26.01 ± 0.00	17.45 ± 0.00	24.5 ± 0.00	16.7 ± 0.00	9.04E+3 $\pm$ 41.13	39.16 ± 0.37	$\mathbf{13.72 \pm 0.22}$
3	26.27 ± 0.00	26.65 ± 0.00	30.0 ± 0.00	61.7 ± 0.00	1.82E+4 $\pm$ 37.23	48.59 ± 0.56	$\mathbf{22.09 \pm 0.29}$
4	27.62 ± 0.00	34.10 ± 0.00	41.3 ± 0.00	74.6 ± 0.00	1.77E+4 $\pm$ 52.30	79.20 ± 1.06	$\mathbf{29.90 \pm 0.47}$
5	44.82 ± 0.00	40.34 ± 0.00	49.0 ± 0.00	85.6 ± 0.00	2.58E+4 $\pm$ 63.54	91.65 ± 1.37	$\mathbf{38.01 \pm 0.52}$
6	$\mathbf{54.48 \pm 0.00}$	46.63 ± 0.00	58.8 ± 0.00	92.5 ± 0.00	4.02E+4 $\pm$ 349.64	526.57 ± 8.74	46.80 ± 0.47
7	70.25 ± 0.00	77.93 ± 0.00	68.0 ± 0.00	102.1 ± 0.00	5.78E+4 $\pm$ 116.92	352.02 ± 6.68	$\mathbf{55.51 \pm 0.57}$
8	70.32 ± 0.00	72.96 ± 0.00	77.7 ± 0.00	103.3 ± 0.00	4.79E+4 $\pm$ 208.70	1332.68 ± 7.82	$\mathbf{63.15 \pm 0.70}$
9	$\mathbf{65.80 \pm 0.00}$	77.34 ± 0.00	84.3 ± 0.00	106.7 ± 0.00	6.54E+4 $\pm$ 491.49	1574.86 ± 9.34	70.30 ± 0.86
10	81.35 ± 0.00	$\mathbf{82.00 \pm 0.00}$	92.7 ± 0.00	120.9 ± 0.00	8.11E+4 $\pm$ 355.34	1876.54 ± 89.20	80.36 ± 0.79

Table 7: Reentrant-2[Hyper]

Net	$c\mu$	MW	MP	FP	PPO	PPO BC	PPO WC
2	28.43 ± 1.37	$\mathbf{22.82 \pm 1.89}$	58.63 ± 2.26	39.75 ± 7.29	9.75E+3 $\pm$ 59.58	66.87 ± 1.11	30.67 ± 0.83
3	41.46 ± 2.14	$\mathbf{36.97 \pm 2.69}$	67.14 ± 2.94	55.52 ± 9.38	2.05E+4 $\pm$ 132.63	482.64 ± 17.98	45.66 ± 0.84
4	72.87 ± 2.68	92.73 ± 3.68	92.96 ± 4.01	72.09 ± 14.37	1.93E+4 $\pm$ 62.07	149.23 ± 3.17	$\mathbf{61.09 \pm 1.30}$
5	$\mathbf{58.65 \pm 2.66}$	72.48 ± 3.92	109.03 ± 5.35	84.17 ± 18.17	2.56E+4 $\pm$ 84.23	371.65 ± 10.81	77.98 ± 1.73
6	$\mathbf{85.68 \pm 5.07}$	133.80 ± 4.32	123.60 ± 5.25	99.51 ± 15.48	6.71E+4 $\pm$ 362.68	1363.93 ± 20.21	93.84 ± 1.26
7	$\mathbf{100.24 \pm 5.96}$	120.23 ± 5.06	135.58 ± 5.99	118.91 ± 15.67	6.54E+4 $\pm$ 214.22	2317.88 ± 15.54	110.48 ± 1.63

C Additional Simulator Design Details

We present a queuing system testing framework. The main goals of the framework are: 1. Provide benchmarks for queuing algorithms 2. Easy to test and deploy with an OpenAI Gym Interface 3. Allow easy configuration of new custom queuing systems with a large degree of freedom

Our testing framework allows the following user interactions with intuitive interface

- Defining a new queueing system or using one provided by our benchmark.
- Defining a policy that takes in observations and output queue priority prediction
- Simulating queueing system trajectories with selected policies

We will detail each of these components of our framework below

C.1 Define queuing system

Ingredients of a queueing system Our framework allows for flexible definition of queuing systems in a straightforward interface. A queuing system can be defined with the following descriptions:

- **Network matrix**: a binary matrix that specifies which server can serve which queue
- **Network transition matrix**: what happens when a server finishes serving a job
- **Service rate matrix**: a matrix that specifies how fast a server can serve a queue. Time of service is drawn from a distribution specified by user using service rate matrix as parameter
- **Arrival rate of queues**: User can define arbitrary arrival pattern for queues as a Python function that inputs time and outputs time until next arrival. This feature allows simulation of time-varying arrivals. User can define arrival rate as a random distribution
- **Queue holding cost**: holding cost per unit of time for each job in each queue
- **Server pool**: We also allow user to define each class of server as a pool of server. When having many servers with the same characteristics, instead of creating many separate servers and inflating the size of network matrix, we allow users to specify a server pool number for each server class. This mechanism allows simulation of large-scale system without slowing the simulation.

Users can define these elements of a queuing sytems in a `.yaml` file and a `.py` file.

Here, we show an example `.yaml` file for configuring a criss-cross network:

```
name: 'criss_cross_bh'
lam_type: 'constant'
lam_params: {val: [0.9, .000001, 0.9]}
network: [[1,0,1],[0,1,0]]
mu: [[2,0,2],[0,1,0]]
h: [1,1,1]
init_queues: [0, 0, 0]
queue_event_options: [[1., 0, 0.],
                      [0., 0., 0.],
                      [0., 0, 1.],
                      [-1., 1., 0.],
                      [0., -1., 0.],
                      [0., 0., -1.],]
```

C.2 Defining a policy

User can define a policy as a function that takes in queue length as observation and output a matrix that represents the policy's prediction of service priority. The matrix has the the same shape as network matrix (# server $\times$ #queues) that assigns priority to each server-queue pair. A policy can be either a static policy that decide priority based on observation with heuristics or contain a neural model to be trained.

https://doi.org/10.52202/079017-2934

C.3 Simulator design

The simulator environment is structured as OpenAI Gym environment. We follow the design of the OpenAI Gym so that users can easily train and test a variety of reinforcement learning algorithms. Each simulation trajectory consists of a sequence of steps (defined in OpenAI Gym `step` format). For each simulation trajectory, the simulator maintain a number of information as states.

C.3.1 Simulator features

We highlight some important features of our simulator below:

- **Job-level tracking** The simulator tracks the states on the job level. We track service time for each job in a queue. At each step, we allocate to decide which job is being served on an individual job level. This mechanism makes it possible for multiple servers to serve a single queue and allows the modeling of parallel server systems.
- **Event-based simulation** Our simulator is event-based. Each step corresponds to one event: arrival in a queue or one job finishes being served. Prior works designed simulators with fixed time-interval for each step. In comparison, we can simulate trajectories with more uneven event intervals with higher speed by reducing wasting steps on intervals without any event. We also allow more precise time keeping.
- **Batch simulation** Our simulator allows simulation of multiple runs in parallel. Our parallelization implementation allow users to leverage accelerators like GPUs to accelerate simulations.

C.3.2 States

In each trajectory, the simulator keeps track a number of variables as states

Based on the elements of queue systems defined above, the simulator also has the capability of drawing a new service duration for a job and arrival duration for a queue. In addition, during a simulation run, the environment keeps track of

- **Service time** Time until service finishes for a job
- **Arrival time** Time until next arrival occurs for a queue
- **Queue length** Length of each queue

At each step, the simulator updates service time and arrival time based on the event duration of the step. The simulator also has the capability to generate service time and arrival time for new jobs based on user specification of the queuing system. The simulator also updates queue lengths at each step based on the event occurred during the step.

C.3.3 An event-based simulation step

At each step, the simulator takes in action represented by the service priority matrix and returns the updated states in OpenAI Gym `step` function format. To simulate a step and obtain the output of the `step function`, our simulator decides an event that occurs based on the following procedure

- **Converting service priority to action matrix** The step function takes in service priority prediction from the policy. The priority matrix can be a float matrix. The `step` function converts this priority matrix into an action matrix that specifies which servers should serve which queues. Users can customize how the assignment is done. Default implementation provides linear assignment, softmax, and Sinkhorn assignment.
- **Job-server allocation** The action matrix pairs server and queues. Our simulator then assigns each job in in a queue to servers that the action matrix decides to serve the queue through an `allocator` function. User can customized how the allocation is performed. The default allocator implementation selects the fastest serving servers and pair them with jobs with shortest service time remaining.
- **Select event** Based on the remaining service time for each job and remaining arrival time for each queue, the simulator decides the closest next event to be either (1) a job finishes being served or (2) a new job arrives for a queue.

- **Update states** Based on the event occurred, the simulator updates the states correspondingly. If a job finishes being served, the simulator removes the job from the queue. If the transition matrix specifies that the job in one queue goes to another queue after being served, an new job is created for the queue that the job transitions into. If a new arrival occurs, the simulator creates a new job for the queue and generate the new arrival time until next arrival in the queue. Finally, the simulator deducts the event duration from all service times and arrival times.

D Additional Experiment Details

D.1 Computational Resources

We run all our experments on an AMD EPYC 7513 32-Core Processor.

https://doi.org/10.52202/079017-2934

Checklist

The checklist follows the references. Please read the checklist guidelines carefully for information on how to answer these questions. For each question, change the default **[TODO]** to [Yes] , [No] , or [N/A] . You are strongly encouraged to include a **justification to your answer**, either by referencing the appropriate section of your paper or providing a brief inline description. For example:

- Did you include the license to the code and datasets? [Yes] See Section **??**.
- Did you include the license to the code and datasets? [No] The code and the data are proprietary.
- Did you include the license to the code and datasets? [N/A]

Please do not modify the questions and only use the provided macros for your answers. Note that the Checklist section does not count towards the page limit. In your paper, please delete this instructions block and only keep the Checklist section heading above along with the questions/answers below.

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] We described our contribution of a new simulating and benchmarking framework for queuing systems, which we detailed in the paper.
   (b) Did you describe the limitations of your work? [Yes] We discussed the limitation of the policies benchmarked and stated that there are plenty of rooms for improvement in section 4.
   (c) Did you discuss any potential negative societal impacts of your work? [N/A] Our queuing simulation framework doesn't pose potential negative societal impact.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes] We have read and followed the guidelines.
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A] We do not include any theoretical result.
   (b) Did you include complete proofs of all theoretical results? [N/A] We do not include any theoretical result.
3. If you ran experiments (e.g. for benchmarks)...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] We provide url to our code release in abstract, which readers can easily use our simulator to benchmark provided methods and additional methods.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] We describe our experimental setting in Section 4 of our paper.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] We provided standard deviation in all tables in our paper.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] We described computational setup in supplementary materials section D.1
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [N/A] We do not use existing assets.
   (b) Did you mention the license of the assets? [N/A] We do not use existing assets.
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes] We provide URL to our code release in abstract.
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A] We do not obtain data from other people.
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A] We do not collect these information.

5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A] We do not use human subjects
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A] We do not use human subjects
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A] We do not use human subjects

https://doi.org/10.52202/079017-2934