Mitigating Metastable Failures in Distributed Systems with Offline Reinforcement Learning

Yueying Li[1,4], Daochen Zha[2], Tianjun Zhang[3], G. Edward Suh[1], Christina Delimitrou[4], Francis Y. Yan[5]
[1]Cornell University [2]Rice University [3]University of California, Berkeley [4]MIT [5]Microsoft Research

Abstract

This paper introduces a load shedding mechanism that mitigates metastable failures through offline reinforcement learning (RL). Previous studies have heavily focused on heuristics that are reactive and limited in generalization, while online RL algorithms face challenges in accurately simulating system dynamics and acquiring data with sufficient coverage. In contrast, our algorithm leverages offline RL to learn directly from existing log data. Through extensive empirical experiments, we demonstrate that our algorithm outperforms rule-based methods and supervised learning algorithms in a proactive, adaptive, generalizable, and safe manner. Deployed in a Java compute service with diverse execution times and configurations, our algorithm exhibits faster reaction time and attains the Pareto frontier between throughput and tail latency.

1 Introduction

Building reliable cloud services is of paramount importance in distributed systems. With the proliferation of microservices (Gan et al., 2019a), it becomes critical for applications to fortify their services against cascading failures and enduring latency degradation (Gan et al., 2019b; 2021). The microservice architecture is particularly vulnerable to a new form of failure known as *metastable failures* (Huang et al., 2022; Bronson et al., 2021), which exhibit a distinct trait of sustaining effects that hold systems in a degraded state and impede their recovery. Notably, these metastable failures have emerged as the culprits behind significant outages at large internet companies.

We focus on rate limiting to protect systems from such catastrophic failures. Prior work has adopted heuristic-based rate limiting to prevent system overload (Netflix; Kumar; Amazon), but these approaches are reactive, causing the system to endure prolonged capacity degradation induced by metastable failures. Additionally, some strategies suffer from convergence issues in non-stationary environments (Figure 1a)[1]. Lastly, these heuristics require configuring a myriad of system-dependent parameters, thus lacking the ability to generalize across different system contexts.

To address these limitations, we explore learning-based approaches for load control to prevent system metastability. An intuitive solution is to use reinforcement learning (RL), as this problem can be naturally framed as a sequential decision process where the rate limit can be predicted based on the system status at each interval. However, existing online RL algorithms are ill-suited to prevent or mitigate metastable failures in the real world due to the following reasons: *1)* it is difficult to access a high-fidelity simulator that accurately captures the system dynamics; *2)* it is infeasible to explore online and collect *unsafe* data in a real cloud system. To overcome these challenges, we ask: “Is it possible to train a load-shedding policy using only existing log data from cloud services, eliminating the need for extensive tuning of static thresholds for rate limiters?”

Offline reinforcement learning (Levine et al., 2020) emerges as an appealing solution for load control. In this paper, we propose *PolicyShedder*, a load shedding policy designed to proactively mitigate metastable failures using offline RL. PolicyShedder learns directly from native system log data, incurring minimal system overhead. By deploying it on a Java compute service that is prone

[1]In this figure, we show a typical case where the heuristic-based load shedder fails to react to system state changes in a timely manner, resulting in cyclic latency spikes and service level objective (SLO) violations.

to metastable failures due to design anti-patterns, we demonstrate that PolicyShedder consistently attains the Pareto frontier between throughput and tail latency, compared with carefully selected heuristics across different contexts (Figure 4), while achieving a 12% lower reaction time, i.e., the time taken to recover from a vulnerable system state with high latency.

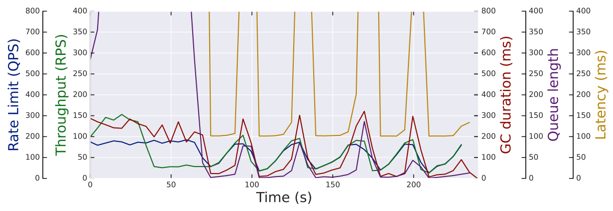
2 Design of PolicyShedder

We model the task of load shedding, aimed at mitigating system overload and metastable failures, as a Markov Decision Process (MDP). The formulation is as follows.

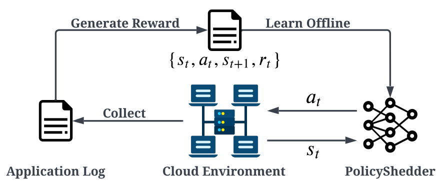
Action space. The objective of PolicyShedder is to generate a rate limit λ_t (number of requests to admit per second) for each pre-configured monitoring time window ΔT. In order to generalize across applications with different service time distributions, we scale the rate limit by Little's law and train an RL model to output $a_t = \lambda_t T_{avg}$, where T_{avg} denotes the average execution time of requests (excluding waiting time in the queue, which can be computed from log data). During deployment, we divide the model's output by T_{avg} to obtain the rate limit λ_t.

State space. The input to PolicyShedder is $s_t = \{\mathrm{Qlen}_t, \mathtt{ewma}(\mathrm{Qlen}_t), \mathrm{Lat}_t, \mathtt{ewma}(\mathrm{Lat}_t)\}$. Here, ewma represents an exponentially weighted moving average over a time interval $[\min\{0, t - n\Delta T\}, t]$, where n is a predefined parameter that captures dependencies in the series of queue lengths (Qlen) and request latencies (Lat) in the log data. An alternative approach involves concatenating a window of historical data into the current observed features. However, there is a trade-off between the state's dimension and the ability to capture dependencies in the time-series data.

Reward. We begin with a well-studied metric in the context of congestion control known as Power, defined as the ratio of throughput to delay (Giessler et al., 1978). Gail and Kleinrock proved that the optimal operation point for the network and individual flows is simultaneously attained when Power is maximized (Gail & Kleinrock, 1981). Considering the importance of tail latency in capturing application SLO requirements and metastable signals, we define the reward using aggregated statistics—average throughput and the 95th percentile of tail latency—over a consecutive 10-second period into the future. This approach helps mitigate the impact of temporal load spikes and captures the delayed effect of rewards. Formally, the reward $R_t(\alpha) = \mathrm{Throughput}_{avg}[t : t + n\Delta T] - \alpha \cdot \mathrm{Latency}_{95}[t : t + n\Delta T]$, where α controls the trade-off between throughput and latency. Detailed experimental setups and results are shown in the appendix (A.4–A.9).



(a) Metastable failures



(b) Overview of PolicyShedder

Figure 1: (a) Illustration of metastable failures. (b) System overview of PolicyShedder. We collect log data from the application and process them using a user-defined function to generate rewards, which are then incorporated into trajectories for training PolicyShedder offline. Once the training completes, PolicyShedder is deployed online to interact with the cloud environment.

3 Conclusion and Outlook

We present a practical application of offline reinforcement learning (RL) in cloud systems, with a focus on mitigating critical system failures. This data-driven offline RL-based abstraction provides a valuable tool for constructing intelligent and reliable distributed systems. An extension of our research is to use a multi-agent formulation to enable load shedding for interconnected cloud services with different user requirements, thereby protecting applications from cascading failures.

Acknowledgments

We thank Zhengyao Jiang for the helpful discussions, especially for providing suggestions on offline RL training methodologies. We also thank Aleksey Charapko for the insightful discussion regarding metastable failures.

URM Statement

The authors acknowledge that at least one key author of this work meets the URM criteria of ICLR 2023 Tiny Papers Track. In this work, Yueying Li meets this criterion.

References

Handling overload. URL: https://ferd.ca/handling-overload.html.

Amazon. Amazon: Using load shedding to avoid overload. URL: https://aws.amazon.com/builders-library/using-load-shedding-to-avoid-overload.

Michael Bain and Claude Sommut. A framework for behavioural cloning. *Machine Intelligence*, 1999.

Lawrence S. Brakmo, Sean W O'Malley, and Larry L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proceedings of the conference on Communications architectures, protocols and applications*, pp. 24–35, 1994.

Nathan Bronson, Abutalib Aghayev, Aleksey Charapko, and Timothy Zhu. Metastable failures in distributed systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, pp. 221–227, New York, NY, USA, 2021. Association for Computing Machinery.

Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time. *Queue*, 14(5):20–53, 2016.

Jonathan D. Chang, Masatoshi Uehara, Dhruv Sreenivas, Rahul Kidambi, and Wen Sun. Mitigating covariate shift in imitation learning via offline data without great coverage. *CoRR*, abs/2106.03207, 2021. URL: https://arxiv.org/abs/2106.03207.

Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Michael Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling, 2021.

Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. Pcc vivace: Online-learning congestion control. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pp. 343–356, 2018.

Scott Fujimoto and Shixiang Shane Gu. A minimalist approach to offline reinforcement learning, 2021.

Scott Fujimoto, Edoardo Conti, Mohammad Ghavamzadeh, and Joelle Pineau. Benchmarking batch deep reinforcement learning algorithms, 2019a.

Scott Fujimoto, David Meger, and Doina Precup. Off-policy deep reinforcement learning without exploration. In *International conference on machine learning*, pp. 2052–2062. PMLR, 2019b.

R. Gail and L. Kleinrock. An invariant property of computer network power. In *Proceedings of the International Conference on Communications*, pp. 63.1.1–63.1.5, Denver, Colorado, June 14-18, 1981.

Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 3–18, 2019a.

Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 19–33, 2019b.

Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. Sage: Practical and scalable ml-driven performance debugging in microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pp. 135–151, New York, NY, USA, 2021. Association for Computing Machinery.

Alfred Giessler, J. D. Haenle, Andreas König, and E. Pade. Free buffer allocation - an investigation by simulation. *Comput. Networks*, 2:191–208, 1978.

Google. Google optimize. URL: https://marketingplatform.google.com/about/optimize/.

Mor Harchol-Balter. Performance modeling and design of computer systems: Queueing theory in action. URL: www.cs.cmu.edu/~harchol/PerformanceModeling/book.html.

Zhang-Wei Hong, Pulkit Agrawal, Remi Tachet des Combes, and Romain Laroche. Harnessing mixed offline reinforcement learning datasets via trajectory weighting. In *The Eleventh International Conference on Learning Representations*, 2023.

Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. Metastable failures in the wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 73–90, Carlsbad, CA, July 2022. USENIX Association.

Michael Janner, Justin Fu, Marvin Zhang, and Sergey Levine. When to trust your model: Model-based policy optimization, 2019.

Nathan Jay, Noga H Rotman, P Brighten Godfrey, Michael Schapira, and Aviv Tamar. A deep reinforcement learning perspective on internet congestion control. URL: https://github.com/PCCproject/PCC-RL.

Zhengyao Jiang, Tianjun Zhang, Michael Janner, Yueying Li, Tim Rocktäschel, Edward Grefenstette, and Yuandong Tian. Efficient planning in a compact latent action space, 2022.

Rahul Kidambi, Aravind Rajeswaran, Praneeth Netrapalli, and Thorsten Joachims. Morel: Model-based offline reinforcement learning, 2020.

Ilya Kostrikov, Rob Fergus, Jonathan Tompson, and Ofir Nachum. Offline reinforcement learning with fisher divergence critic regularization. In *International Conference on Machine Learning*, pp. 5774–5783. PMLR, 2021a.

Ilya Kostrikov, Ashvin Nair, and Sergey Levine. Offline reinforcement learning with implicit q-learning. *CoRR*, abs/2110.06169, 2021b. URL: https://arxiv.org/abs/2110.06169.

Aviral Kumar, Aurick Zhou, George Tucker, and Sergey Levine. Conservative q-learning for offline reinforcement learning, 2020.

Vikas Kumar. Load shedding with NGINX using adaptive concurrency control. URL: https://tech.olx.com/load-shedding-with-nginx-using-adaptive-concurrency-control-part-1-e59c7da6a6df.

Kwei-Herng Lai, Daochen Zha, Junjie Xu, Yue Zhao, Guanchu Wang, and Xia Hu. Revisiting time series outlier detection: Definitions and benchmarks. In *Thirty-fifth conference on neural information processing systems datasets and benchmarks track (round 1)*.

Kwei-Herng Lai, Daochen Zha, Guanchu Wang, Junjie Xu, Yue Zhao, Devesh Kumar, Yile Chen, Purav Zumkhawaka, Minyang Wan, Diego Martinez, et al. Tods: An automated time series outlier detection system. In *Proceedings of the aaai conference on artificial intelligence*, volume 35, pp. 16060–16062, 2021.

Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. Offline reinforcement learning: Tutorial, review, and perspectives on open problems, 2020.

Yuening Li, Daochen Zha, Praveen Venugopal, Na Zou, and Xia Hu. Pyodds: An end-to-end outlier detection system with automated machine learning. In *Companion Proceedings of the Web Conference 2020*, pp. 153–157, 2020.

Yuening Li, Zhengzhang Chen, Daochen Zha, Kaixiong Zhou, Haifeng Jin, Haifeng Chen, and Xia Hu. Automated anomaly detection via curiosity-guided search and self-imitation learning. *IEEE Transactions on Neural Networks and Learning Systems*, 33(6):2365–2377, 2021a.

Yuening Li, Zhengzhang Chen, Daochen Zha, Kaixiong Zhou, Haifeng Jin, Haifeng Chen, and Xia Hu. Autood: Neural architecture search for outlier detection. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pp. 2117–2122. IEEE, 2021b.

Radhika Mittal, Terry Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based congestion control for the datacenter. In *Sigcomm '15*, 2015.

Netflix. Concurrency limiter's heuristic algorithm. URL: https://github.com/Netflix/concurrency-limits/blob/master/concurrency-limits-core/src/main/java/com/netflix/concurrency/limits/limit/Gradient2Limit.java#L76-L84.

Samarth Sinha, Homanga Bharadhwaj, Aravind Srinivas, and Animesh Garg. D2RL: Deep dense architectures in reinforcement learning. *arXiv preprint arXiv:2010.09163*, 2020.

Ziyu Wang, Alexander Novikov, Konrad Zolna, Josh S Merel, Jost Tobias Springenberg, Scott E Reed, Bobak Shahriari, Noah Siegel, Caglar Gulcehre, Nicolas Heess, et al. Critic regularized regression, 2020.

Keith Winstein and Hari Balakrishnan. TCP ex Machina: Computer-generated congestion control. *ACM SIGCOMM Computer Communication Review*, 43(4):123–134, 2013.

Francis Y. Yan, Jestin Ma, Greg D. Hill, Deepti Raghavan, Riad S. Wahby, Philip Levis, and Keith Winstein. Pantheon: the training ground for internet congestion-control research. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 731–743, Boston, MA, July 2018. USENIX Association.

Tianhe Yu, Garrett Thomas, Lantao Yu, Stefano Ermon, James Y. Zou, Sergey Levine, Chelsea Finn, and Tengyu Ma. MOPO: Model-based offline policy optimization, 2020.

Tianhe Yu, Aviral Kumar, Rafael Rafailov, Aravind Rajeswaran, Sergey Levine, and Chelsea Finn. Combo: Conservative offline model-based policy optimization, 2021.

Daochen Zha, Kwei-Herng Lai, Mingyang Wan, and Xia Hu. Meta-AAD: Active anomaly detection with deep reinforcement learning. In *2020 IEEE International Conference on Data Mining (ICDM)*, pp. 771–780. IEEE, 2020.

Daochen Zha, Louis Feng, Bhargav Bhushanam, Dhruv Choudhary, Jade Nie, Yuandong Tian, Jay Chae, Yinbin Ma, Arun Kejariwal, and Xia Hu. Autoshard: Automated embedding table sharding for recommender systems. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 4461–4471, 2022a.

Daochen Zha, Louis Feng, Qiaoyu Tan, Zirui Liu, Kwei-Herng Lai, Bhargav Bhushanam, Yuandong Tian, Arun Kejariwal, and Xia Hu. Dreamshard: Generalizable embedding table placement for recommender systems. In *Advances in Neural Information Processing Systems*, 2022b.

Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G. Edward Suh, and Christina Delimitrou. Sinan: ML-based and QoS-aware resource management for cloud microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pp. 167–181, New York, NY, USA, 2021. Association for Computing Machinery.

A Appendix

A.1 Related Work

A.1.1 Offline RL

In offline RL, a policy is learned from logged data, collected from an environment over a period of time, interaction with the environment is not required. The policy used affects the data distribution collected from an environment. When a policy is learned using an offline dataset, the data distribution when the learned policy is in use differs from the logged data, resulting in a data distribution shift. This remains the fundamental problem with offline RL and several different approaches have been proposed to tackle it.

Offline RL methods can be grouped into two categories in terms of learning and utilizing a model of the environment. Model-based offline RL methods (Janner et al., 2019; Kidambi et al., 2020; Yu et al., 2020; 2021; Jiang et al., 2022; Zha et al., 2022b;a) train a model of the environment using state-action transitions from the logged data. These methods utilize the learned model to generate synthetic episodes, controlled by the policy being trained. The policy parameters are updated using a combination of real episodes (from the logged data) and synthetic ones until convergence. On the other hand, model-free methods (Fujimoto et al., 2019a;b; Kumar et al., 2020; Wang et al., 2020; Kostrikov et al., 2021a) learn a policy that maps states to actions to maximize returns directly. Our methods directly leverage IQL but change the action partition and normalization to be able to generalize to different contexts.

A.1.2 Performance Debugging, Anomaly Detection, and Root Cause Analysis

Anomaly detection has been widely studied in machine learning Zha et al. (2020); Lai et al. (2021); Lai et al.; Li et al. (2021a;b; 2020). Recently, anomaly detection has been applied to performance debugging in cloud services (Gan et al., 2021; 2019b; Zhang et al., 2021). Sage uses unsupervised learning and Causal Bayesian Networks for modeling causal relationships among microservices and uses counterfactuals to detect root causes (services and resources) of latency service-level objective (SLO) violation. 93% accuracy in correctly identifying the root cause of QoS violations.

Our problem of mitigating metastable failure is related to but different from performance debugging or mitigating SLO violation. They are related because learning-based methods have great potential to improve the reliability of distributed systems by learning through history logs/traces/metrics. Instead of coding hard-wired mapping of predefined signals/events for an anomaly to mitigation actions (like restarting servers, rebooting, or adding more resources through auto-scaling), the adaptation of anomaly detection and mitigation to new contexts can become an automatic procedure done by machines in a few days with retraining.

They are also different. Performance debugging for SLO violations can be caused by the contention of resources and can be mitigated by resource isolation or auto-scaling. For a metastable failure, it is characterized by a sustaining effect loop either by capacity degradation or workload amplification, which complicates the mitigation strategies.

A.2 Background

We first discussed how we arrive at the right formulation and abstraction. Our problem boils down to setting the right rate or concurrency limit for a service, according to the observed service status (latency, queue size, etc.) When enabled, our limiter will reject excess RPS (request per second) to allow instances to run at a safe and stable state. Our goal is to maximize the throughput and minimize the tail latency of the service and prevent metastable failures from happening when the system enters a vulnerable state[2].

In order to find the right limit of cloud service at the application level, traditionally, people draw wisdom from queuing theory (Harchol-Balter) and manually configured fixed concurrency limits measured via a process of performance testing and profiling. While this provided an accurate value at

[2]Extended discussion on how metastable state and vulnerable state are defined can be found in Huang et al. (2022).

that moment in time, the measured limit would quickly become stale as a system's topology changes due to partial outages, auto-scaling, or from code push that impact latency characteristics (url).

Adaptive rule-based approach: A natural solution is to use an adaptive rate limiter, or equivalently, a concurrency limiter. An industry example of an adaptive concurrency limiter is from Netflix (Netflix), which draws inspiration from TCP congestion control algorithms (Cardwell et al., 2016; Brakmo et al., 1994; Winstein & Balakrishnan, 2013; Yan et al., 2018), that seek to determine how many packets may be transmitted at a time without incurring timeouts or increased latency. These algorithms, when deployed on the server side, are based on the assumption that latencies are good proxies for queuing. However, when the system is going under a metastable state - each request in the queue could take longer to execute; and moreover, the latency distributions of services could be drastically different. Hence these methods are neither accurate enough to capture system state changes nor generalizable enough to unseen system conditions to prevent or to mitigate metastable failures.

Adaptive online learning/reinforcement learning: Similarly, people can use online learning to dynamically adjust the load based on real-time feedback, according to a learned policy (Jay et al.). As an example, using multi-armed bandit or implementing more sophisticated RL-based adaptive online learning to take more states into account has shown promising performance in network congestion control problem (Dong et al., 2018; Mittal et al., 2015). Online learning has shown some promise in other applications too (Google), however, to implement fully online learning algorithms in the real world, it is necessary to collect responses and update configurations in near real-time, which poses significant challenges to the infrastructure. Non-Bayesian online algorithms tend to explore extensively in the initial rounds. This can have a major impact on user experience and lead to SLO violation before the algorithm converges. Furthermore, since the environment is non-stationary, the algorithm may be consistently in the exploration phase, which may cause convergence issues.

Offline (un-/semi-/supervised) learning: The third option is to learn a policy from log data without expensive online exploration with a supervised learning approach like behavior cloning (BC) (Bain & Sommut, 1999). Supervised learning is suitable if we can learn a mapping from the state of the system, load shedding action to the utility function of predicted latency and throughput of the service. It is used in congestion control literature. However, this is untenable due to the large state space across different services. Furthermore, inaccurate predictions can cause a feedback cycle known as cascading errors in the long term (Chang et al., 2021). In our problem setup, if we have a slight prediction error that predicts a sub-optimal action, errors can be compounded and lead to more unstable failures.

A.3 Preliminaries for Offline RL

The RL problem is formulated in the context of a Markov decision process (MDP) $(\mathcal{S}, \mathcal{A}, p_0(s), p(s'|s,a), r(s,a), \gamma)$, where $\mathcal{S}$ is a state space, $\mathcal{A}$ is an action space, $p_0(s)$ is a distribution of initial states, $p(s'|s,a)$ is the environment dynamics, $r(s,a)$ is a reward function, and γ is a discount factor. The agent interacts with the MDP according to a policy $\pi(a|s)$. The goal is to obtain a policy that maximizes the cumulative discounted returns:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) | s_0 \sim p_0(\cdot), a_t \sim \pi(\cdot|s_t), s_{t+1} \sim p(\cdot|s_t, a_t)\right].$$

Off-policy RL methods based on approximate dynamic programming typically utilize a state-action value function (Q-function), referred to as $Q(s,a)$, which corresponds to the discounted returns obtained by starting from the state s and action a, and then following the policy π.

Offline reinforcement learning with implicit Q-learning. In contrast to online (on-policy or off-policy) RL methods, offline RL uses previously collected data without any additional data collection. Like many recent offline RL methods, our work builds on approximate dynamic programming methods that minimize temporal difference error, according to the following loss:

$$L_{TD}(\theta) = \mathbb{E}_{(s,a,s')\sim\mathcal{D}}[(r(s,a) + \gamma \max_{a'} Q_{\hat{\theta}}(s',a') - Q_{\theta}(s,a))^2], \tag{1}$$

where $\mathcal{D}$ is the dataset, $Q_\theta(s,a)$ is a parameterized Q-function, $Q_{\hat{\theta}}(s,a)$ is a target network (e.g., with soft parameters updates defined via Polyak averaging), and the policy is defined as $\pi(s) = \arg\max_a Q_\theta(s,a)$.

There are three functions to train in IQL:

$$L_V(\psi) = \mathbb{E}_{(s,a)\sim D}[L_2^\tau(Q_\theta(s,a) - V_\psi(s))]$$

where $L_2^\tau(u) = |\tau - \mathbb{1}(u<0)|u^2$.

The Q-function is trained with the state-value function to avoid querying the actions.

$$L_Q(\theta) = \mathbb{E}_{(s,a,r,a')\sim D}[(r + \gamma V_\psi(s') - Q_\theta(s,a))^2]$$

Finally, the policy function is trained by using advantage-weighted regression.

$$L_\pi(\phi) = \mathbb{E}_{(s,a)\sim D}[\exp(\beta(Q_\theta - V_\psi(s)))\log \pi_\phi(a|s)]$$

A.4 Experiments

Table 1: Rewards of PolicyShedder and the baselines with different average execution times.

Method	In-distribution			Out-of-distribution		Metastability
	80 ms	100 ms	120 ms	60 ms	140 ms	all
Best heuristic	41.17 ± 0.87	24.99 ± 0.71	10.47 ± 0.84	31.00 ± 3.41	-21.44 ± 0.92	1/5
BC	19.91 ± 12.87	-1379.50 ± 164.43	-1530.12 ± 95.98	-1312.36 ± 429.97	-4685.18 ± 441.51	2/5
IQL	50.47 ± 1.60	**30.20 ± 1.65**	13.59 ± 1.27	41.06 ± 1.57	-1.68 ± 1.41	2/5
TD3+BC	32.20 ± 1.65	12.34 ± 3.24	-43.56 ± -6.94	-6.12 ± 6.34	9.42 ± 2.83	2/5
DT	24.99 ± 0.71	23.48 ± 3.71	13.36 ± 9.97	-39.12 ± 89	19.34 ± 1.70	1/5
CQL	13.59 ± 1.27	-4.18 ± 0.21	-4.50 ± 2.34	-1.36 ± 0.97	9.19 ± 2.78	3/5
PolicyShedder	**53.05 ± 1.50**	29.14 ± 2.13	**16.59 ± 0.29**	**45.20 ± 1.22**	**32.45 ± 3.21**	0/5

We train PolicyShedder on the log data collected from java application environments with different average execution times and heap sizes, where the execution time of these applications ranges from $\{80, 100, 120\}$ (ms), and the heap size is in $\{192, 256, 512\}$ (MB). The logging policy is a heuristic policy based on TIMELY algorithm (Mittal et al., 2015), which is widely used in datacenter.

The initial version of PolicyShedder is trained with implicit Q-learning (IQL) (Kostrikov et al., 2021b). We used around 500 trajectories with the reward ($\alpha = 0.25$), and each trajectory contains around 4 minutes of log data with monitoring interval as 1 sec. To adapt to sporadic traffic, we choose $\Delta T = \min\{1, \text{time with at least 3 consecutive requests}\}$. However, we found this vanilla offline RL approach is not able to reason about the performance well under distribution shift in transition dynamics (which is the key characteristic of Metastable system, compared to traditional congestion control). Hence, we proposed to use feature normalization and advantage weighted reweighting for our datasets (Hong et al., 2023).

To evaluate the generalizability of PolicyShedder, we test in both in-distribution and out-of-distribution environments. The former uses the same ranges of execution times and the heap sizes, while for the latter, the execution time is selected from $\{60, 140\}$ and only the heap size is selected in the same range. We compare PolicyShedder with several different baselines: **1) Heuristic:** It uses heuristic strategies (Netflix) to control the rate limit in a certain range. [3] We adopt grid-search for the heuristics and report the best result. **2) Behavior cloning (BC) and Offline RL:** BC is an imitation learning algorithm; it uses supervised learning losses to train the policy to imitate the behavioral policies recorded in the log. We include it as it is a common baseline in offline RL research (Kostrikov et al., 2021b; Kumar et al., 2020). We also choose the mostly widely used offline

[3]Originally, we use the load-shedder baseline simply as the one in (Netflix). Note that the concurrency limit can be translated to the rate limit because we know the system queue lengths at each time stamp. However, we found that the heuristic is not able to fully prevent the metastable failure from happening due to the delayed nature of load-shedding actions, and requires some prior knowledge of service concurrency limit. Hence, we improve upon the baseline with a stronger version of concurrency control (Mittal et al., 2015).

RL methods, including one-step, pessimistic, and conservative algorithms. Specifically, we implement Conservative Q-learning (CQL) (Kumar et al., 2020), Implicit Q-learning (IQL) (Kostrikov et al., 2021b), TD3+BC (Fujimoto & Gu, 2021), Decision transformer (DT) (Chen et al., 2021))

Table 1 summarizes the results. **Observation 1:** PolicyShedder significantly outperforms the best heuristic, showing the promise of handling metastable failures with offline RL. **Observation 2:** Behavior cloning delivers unsatisfactory performance. This is because the log contains both good and bad behaviors. The supervised policy may have learned undesirable behaviors from the log.

A.5 HYPERPARAMETERS

We report the hyperparameters used for training the RL agent. Unless otherwise stated, we use grid search for hyperparameter optimization.

Table 2: Hyperparameter of Behavior Cloning (BC).

Hyperparameter	Value
Batch size	100
Regularization factor	0.5

Table 3: Hyperparameter of Implicit Q-Learning (IQL).

Hyperparameter	Value
Actor learning rate	3×10^{-4}
Critic learning rate	3×10^{-4}
Actor optimizer	Adam
Critic optimizer	Adam
Batch size	256
N-step TD calculation	1
Discount factor	0.99
Target network synchronization coefficiency	0.005
The number of Q functions for ensemble	2
The expectile value for value function training	0.7
Inverse temperature value	3.0
The maximum advantage weight value to clip	100.0

Table 4: Hyperparameter of TD3+BC.

Hyperparameter	Value
Actor learning rate	3×10^{-4}
Critic learning rate	3×10^{-4}
Batch size	256
N-step TD calculation	1
Discount factor	0.99
Target network synchronization coefficiency	0.005
The number of Q functions for ensemble	2
Standard deviation for target noise	0.2
Clipping range for target noise	0.5
Alpha	2.5
Interval to update policy function	2

Table 5: Hyperparameter of Conservative Q-Learning (CQL).

Hyperparameter	Value
Actor learning rate	3×10^{-4}
Critic learning rate	3×10^{-4}
Learning rate for temperature parameter of SAC	1×10^{-4}
Learning rate for alpha	1×10^{-4}
Batch size	256
N-step TD calculation	1
Discount factor	0.99
Target network synchronization coefficiency	0.005
The number of Q functions for ensemble	2
Initial temperature value	1.0
Initial alpha value	1.0
Threshold value	10.0
Constant weight to scale conservative loss	5.0
The number of sampled actions to compute	10

A.6 Ablation Studies

Table 6: Model ablation studies.

	Dropout	Hidden	Dense	BatchNorm	Overhead (s)	In-distribution	OOD
MLP - 1	0.5	[32,32]	No	Yes	0.0032	34 ± 1.65	38.5 ± 1.34
MLP - 2	0.5	[16,16]	No	Yes	0.0031	32 ± 2.09	32.4 ± 2.23
MLP - 3	0.5	[8, 8]	No	Yes	0.0031	23.5 ± 1.94	-19.2 ± 2.34
MLP - 4	0.2	[32,32]	No	No	0.0032	26 ± 1.42	23.4 ± 1.22
MLP - 5	0.5	[32,32]	No	No	0.0031	30.5 ± 0.65	33.3 ± 3.88
MLP - 6	0.8	[32,32]	No	No	0.0031	22 ± 1.23	33.2 ± 5.32
MLP - 7	NA	[32,32]	No	No	0.0028	28.25 ± 1.21	31.2 ± 4.20
MLP - 8	0.5	[32,32,32]	No	Yes	0.0035	35.5 ± 1.01	45.3 ± 4.39
MLP - 9	0.5	[32,32,32,32]	No	Yes	0.0036	34.75 ± 0.49	43.5 ± 3.23
MLP - 10	0.5	[32,32]	Yes	Yes	0.0033	36 ± 2.65	49.5 ± 1.32
Linear	0.5	[32]	No	Yes	0.0028	5.75 ± 1.92	-3.5 ± 1.21

We now study how each design choice affects performance. In the figure below (Figure ??), we aggregate the score across the in-distribution and OOD setups.

Feature choices: FDC1: choice of multiple feature inputs including memory and CPU utilization; FDC2: without normalization; FDC3: without EWMA features. Surprisingly, we find that additional features like memory and CPU utilization are not helpful in improving the model performance.

Model choices: MDC1: use DNN model; MDC2: use LSTM model. We observed that although changing the policy network to a more complex LSTM model seems to be able to get us a slightly higher score, the overhead introduced by the additional complexity overshadows the benefit.

Look-ahead interval: We sweep the lookahead window with linear search from $n = 5$ to $n = 15$ but report only three points for the sake of space. The look-ahead window is used for in the reward formulation. In essence, a smaller window makes the system less reactive but may be more sensitive to spurious long requests.

In Table 6, we consider different model architectures, reporting both system overhead and model performance for in distribution and out-of-distribution experiments under different neural architectures. We can see that the overhead is not sensitive to the shape of hidden layers (MLP - 7-9); however, the more the number of layers, the more the agent's performance and sample complexity deteriorates (Sinha et al., 2020). The model performance is sensitive to the dense connections (MLP - 10 vs MLP - 1), especially for OOD environment. Moreover, the wider the hidden layers and the more dropouts, the better the generalizability.

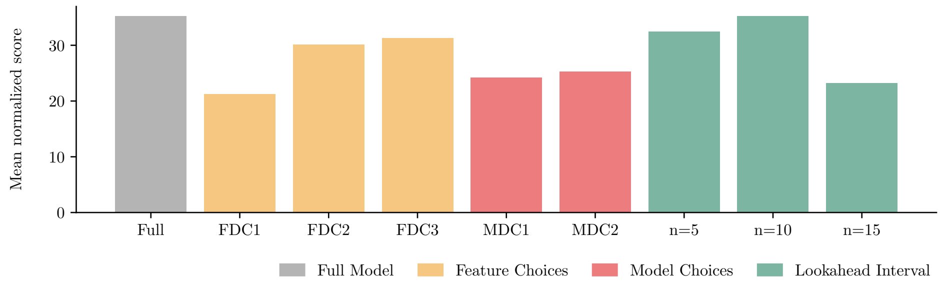


Figure 2: Ablation studies

A.7 Parser Details

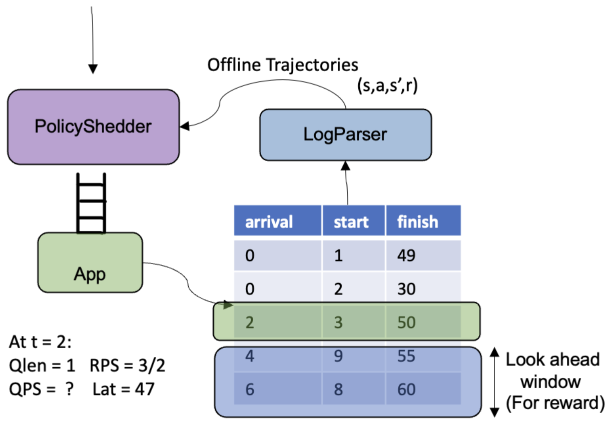


Figure 3: Parser for the log

In Figure 3, we show the details about how we parse the existing request logs to get the trajectories used for offline RL training. The action are indirectly recorded from the log, and the reward can be calculated by taking tail latency and QPS for future look-ahead windows. We further normalize the action and reward according to the service characteristics (i.e. average execution time).

A.8 OOD performance

In this section, we focus on PolicyShedder's performance under the distribution shift. We make the service have a higher / lower average execution time due to the code upgrade, and report its performance across different system setups. We observe that compared with best heuristic (under linear search) and Behavior cloning which simply learns from the heuristics, it is better at reasoning about the right actions under OOD environment.

Table 7: Out-of-distribution rewards on different execution times and heap sizes for the best heuristic (80, 120, 0.75), behavior cloning, and our PolicyShedder. The best reward is highlighted in boldface, and the second best reward is underlined.

Method	Execution time 90			Execution time 140		
	192	256	512	192	256	512
(80, 120, 0.75)	<u>29.65</u>	<u>27.66</u>	<u>35.69</u>	<u>-20.20</u>	<u>-21.75</u>	<u>-22.37</u>
Behavior cloning	-708.82	-1550.01	-1678.25	-5086.85	-4070.34	-4898.35
PolicyShedder	**43.28**	**39.88**	**40.03**	**-1.67**	**0.05**	**-3.40**

A.9 VISUALIZATION

Figure 4 shows that PolicyShedder achieves a better tradeoff between throughput and latency compared with the heuristics., i.e., higher throughput and lower latency across all system configurations. The legend tuple shows the average execution time (ms) of the requests in the workload and garbage collection (GC) heap size (MB) configurations.

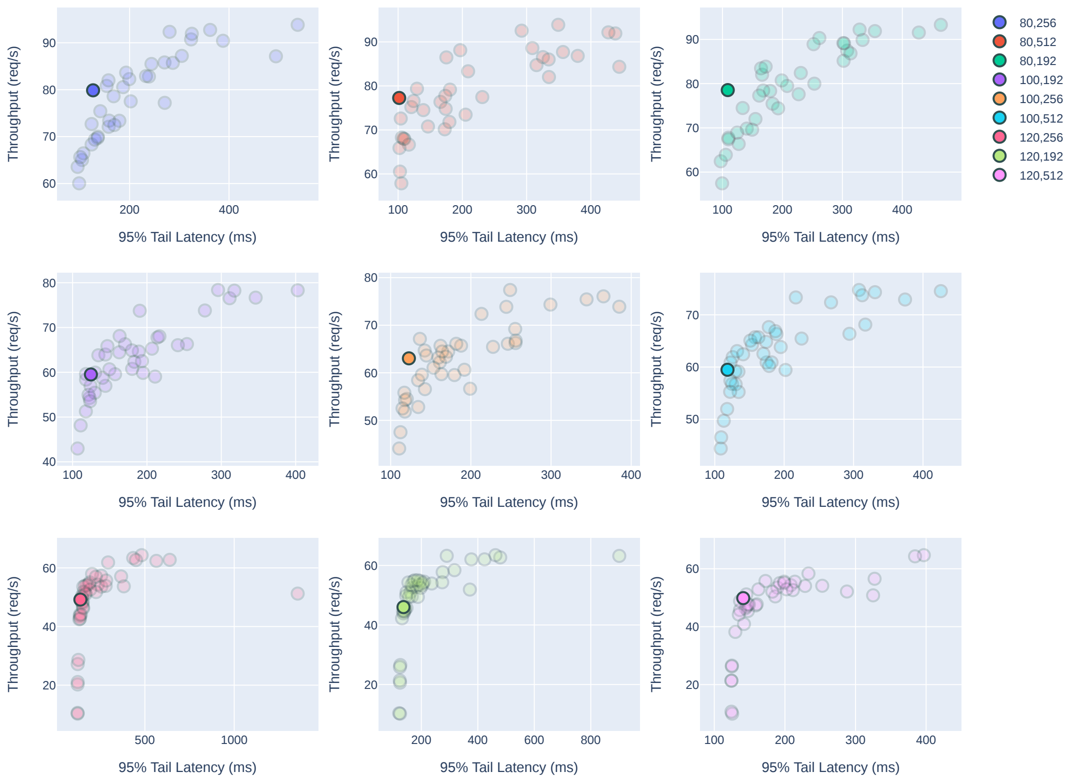
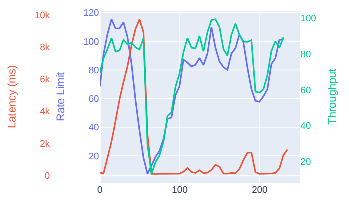
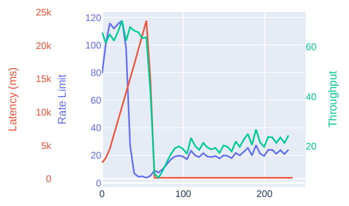


Figure 4: Visualization of PolicyShedder against the heuristics, where PolicyShedder is highlighted as solid selected points.

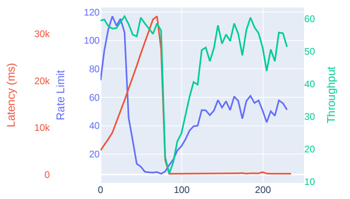
A.10 Case Study



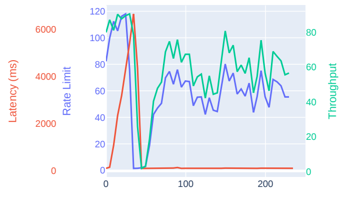
(a) Heuristic (execution 100)



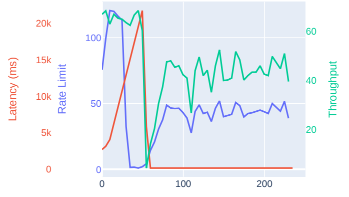
(b) Heuristic (execution 140)



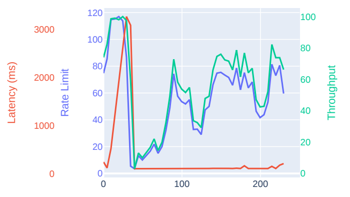
(c) Heuristic (execution 160)



(d) PolicyShedder (execution 100)



(e) PolicyShedder (execution 140)



(f) PolicyShedder (execution 160)

Figure 5: Visualization of the misconfigured heuristic policy when the system has a code upgrade that makes average execution time from 140 ms to 160 ms. The heuristics' reaction time is longer compared with PolicyShedder by average 12%. (a) vs (d) shows how our system is more stable. (b) vs (e) shows a misconfigured heuristic could end up sacrificing the long-term throughput of the service, while our PolicyShedder is less conservative. (c) vs (f) further demonstrate a much faster reaction.