TOWARDS HANDLING METASTABLE FAILURES IN DISTRIBUTED SYSTEMS WITH OFFLINE RL

Anonymous authors
Paper under double-blind review

1 INTRODUCTION

Building reliable cloud services has long been an important area in distributed systems. With the proliferation of microservice (Gan et al., a), it is important for applications to shield their services from cascading failures and sustained latency degradation (Gan et al., b; 2021). The microservice design pattern is prone to a new type of failure, metastable failure (Huang et al., 2022; Bronson et al., 2021), a class of system failures characterized by sustaining effects that keep systems in a degraded state and resist recovery, and were the culprits behind big outages at large internet companies.

Prior works often adopt heuristics to limit the response rate to prevent system overload (Netflix; Kumar; Amazon). However, their solutions are more reactive than proactive; systems that encounter metastability still suffer from long-term capacity degradation. Moreover, such strategy has convergence issues under non-stationary environments (Figure 1a)[1]. Lastly, the heuristics need to configure lots of system-dependent parameters which cannot generalize across different system contexts.

To address these limitations, we explore learning-based approaches for load control to prevent system metastability. One natural solution is to use Reinforcement Learning (RL) since this problem can be naturally treated as a sequential decision process where the rate limit can be predicted based on the system status at each interval. However, there are three key challenges in mitigating metastable failures: 1) It is hard to access a high-fidelity simulator that can accurately capture the dynamics of the system. 2) Building a lightweight interface for different system contexts is challenging. 3) Collecting *unsafe* data in a real cloud system is infeasible.

In this paper, we propose a load-shedder for resolving metastability failures based on offline RL. We learn from native system logging data without expensive feature engineering. Compared to rule-based methods, our offline-RL-based algorithm (PolicyShedder) resolves metastability failures in a proactive, adaptive, generalizable, and safe manner. We deploy our solution on a Java compute service, demonstrating that our policy can always obtain Pareto frontier compared with carefully selected heuristics across different contexts (Figure 2), and react faster by 12% than heuristics.

2 METHOD

We model the task of load-shedding to resolve metastable failure as a Markov Decision Process (MDP). Below is the formulation:

Action Space: The output of our RL-based load shedder policy is a rate limit λ_t (how many requests to admit per second) at each pre-configured time stamp (ΔT). However, in order to be generalizable to different applications with different service time distributions, we scale them by little's law. $a_t = \lambda_t T_{avg}$, where T_{avg} is the average execution time of the requests (excluding waiting time in the queue). In deployment, we divide the action by the T_{avg} to ensure a stable queuing status.

State Space: The input to our RL agent is $s_t = \{\mathrm{Qlen}_t, \texttt{ewma}(\mathrm{Qlen}_t), \mathrm{Lat}_t, \texttt{ewma}(\mathrm{Lat}_t)\}$. Each `ewma` is an exponentially weighted moving average, which captures the dependency of the sequential data. An alternative formulation is to leverage a window of historical data and concatenate it to the current observed features. There is a trade-off between the state dimensions and the ability to incorporate the dependencies in the time-series data.

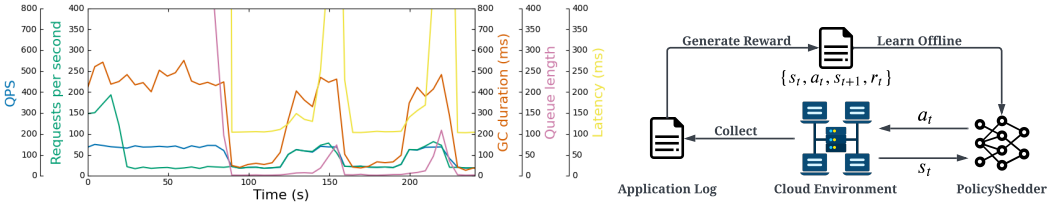
Reward: To choose the reward function, we start from a well-studied metric in the context of congestion control called Power. Power is defined as Power = Throuhput/Delay. Since the reward

[1]In this figure, we show a typical case where the heuristic-based load shedder cannot react to system state changes in a timely manner, resulting in cyclic latency spikes and SLO violations

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

Method	In-distribution			Out-of-distribution	
	80 ms	100 ms	120 ms	60 ms	140 ms
Best heuristic	41.17 $\pm$ 0.87	24.99 $\pm$ 0.71	10.47 $\pm$ 0.84	31.00 $\pm$ 3.41	-21.44 $\pm$ 0.92
Behavior cloning	19.91 $\pm$ 12.87	-1379.50 $\pm$ 164.43	-1530.12 $\pm$ 95.98	-1312.36 $\pm$ 429.97	-4685.18 $\pm$ 441.51
PolicyShedder	**50.47** $\pm$ **1.60**	**30.20** $\pm$ **1.65**	**13.59** $\pm$ **1.27**	**41.06** $\pm$ **1.57**	**-1.68** $\pm$ **1.41**

should be general to different tenants' requirements on their trade-off on tail latency and throughput ($\text{Latency}, \text{Throughput}$) and because tail latency is a more important metric to capture the application's SLO requirement and metastability signals. Leveraging the log data, we choose reward as the aggregated statistics (tail latency in 90% and average throughput) in the future 10 seconds to avoid the temporal load spike's effect. $R(\alpha) = \text{Throughput} - \alpha\text{Latency}$.



(a) Metastable failures

(b) An overview of PolicyShedder

Figure 1: System high-level diagram of PolicyShedder. We collect the data from the application and then parse the log with a user-predefined function to generate rewards and append them to different trajectories. These trajectories are then used to learn a policy offline ("PolicyShedder"), which is further deployed online to interact with the cloud environment.

3 Experiment

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 PolicyShedder is trained with IQL Kostrikov et al. (2021b) and is able to generate new policies from the data to be deployed online. We used around 500 trajectories with the reward ($\alpha = 0.25$), and each trajectory contains around 4 minutes of logging data.

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 two baselines: **1) Heuristic:** It uses heuristic strategies (Netflix) to control the rate limit in a certain range We adopt grid-search for the heuristics and report the best result. **2) Behavior cloning:** It uses supervised learning losses to train the policy to imitate the behavioral policies recorded in the log.

Table 3 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. See the complete results, performance visualizations, and case studies in Appendix A.5, A.6, A.7.

4 Conclusion and Outlook

We demonstrate the real-world use case of offline RL on cloud systems, especially for mitigating critical system failures. This data-driven offline RL-based abstraction can be helpful for building an intelligent and reliable distributed system. An extension to our work is to use multi-agent formulation for connected cloud services to do load-shedding to shield applications from cascaded failures.

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. ISBN 9781450384384. doi: 10.1145/3458336.3465286. URL https://doi.org/10.1145/3458336.3465286.

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.

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, 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.

Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, Christina Delimitrou, Y Zhang, D Cheng, A Shetty, P Rathi, N Katarki, A Bruno, J Hu, B Ritchken, B Jackson, K Hu, M Pancholi, Y He, B Clancy, C Colen, F Wen, C Leung, S Wang, L Zaruvinsky, M Espinosa, R Lin, Z Liu, J Padilla, and C Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud edge systems. a. doi: 10.1145/3297858.3304013. URL https://doi.org/10.1145/3297858.3304013.

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. b. doi: 10.1145/3297858.3304004. URL https://doi.org/10.1145/3297858.3304004.

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. ISBN 9781450383172. doi: 10.1145/3445814.3446700.

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.

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. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/huang-lexiang.

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. a. URL https://github.com/PCCproject/PCC-RL.

Nathan Jay, Noga H Rotman, P Brighten Godfrey, Michael Schapira, and Aviv Tamar. A deep reinforcement learning perspective on internet congestion control. b. 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.

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.

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.

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.

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. ISBN 9781450383172. doi: 10.1145/3445814.3446693. URL https://doi-org.proxy.library.cornell.edu/10.1145/3445814.3446693.

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; b; 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 metastability 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), 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., a). 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; Jay et al., b). 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 logging 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. 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 Implementation Details

We build an environment of a compute-intensive Java microservice and attach a logging service and load-shedding frontend to it. The logging service records the latency (Lat), request per second (RPS), and throughput (QPS) at the time of each request. The load generator is an open-loop generator with a poison arrival rate and uniform service time distribution with small variance.

A.5 Complete Results

Table 2: In-distribution rewards on different execution times and heap sizes for all the heuristics, behavior cloning, and our PolicyShedder. Each heuristic is represented as (T_{min}, T_{max}, β). The best reward is highlighted in boldface, and the second best reward is underlined. We also report the percentage of improvement of PolicyShedder (boldface) over the best baseline (underline).

Method	Execution time 80			Execution time 100			Execution time 120		
	192	256	512	192	256	512	192	256	512
(80, 120, 0.25)	24.92	9.80	11.36	13.84	8.54	9.18	2.06	-41.84	2.02
(80, 120, 0.5)	29.57	42.36	45.05	20.28	14.65	25.58	10.96	11.66	10.54
(80, 120, 0.75)	39.94	41.78	41.81	25.91	24.88	24.17	9.77	9.99	11.65
(60, 140, 0.25)	10.46	13.97	5.20	2.86	5.39	-7.15	-20.91	-6.57	-61.26
(60, 140, 0.5)	40.50	40.15	34.90	6.30	12.67	15.07	10.46	7.74	-0.22
(60, 140, 0.75)	40.29	37.41	34.16	20.95	20.88	15.42	10.73	10.33	3.85
Behavior cloning	7.63	14.41	37.68	-1573.52	-1393.51	-1171.48	-1561.20	-1629.00	-1400.16
PolicyShedder	**51.33**	**48.23**	**51.85**	**28.36**	**32.35**	**29.89**	**11.81**	**14.37**	**14.61**
Improvement	26.74%	13.86%	15.09%	9.46%	30.02%	16.85%	7.76%	23.24%	25.41%

Table 3: 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)	29.65	27.66	35.69	-20.20	-21.75	-22.37
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.6 Visualization

Figure 2 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.

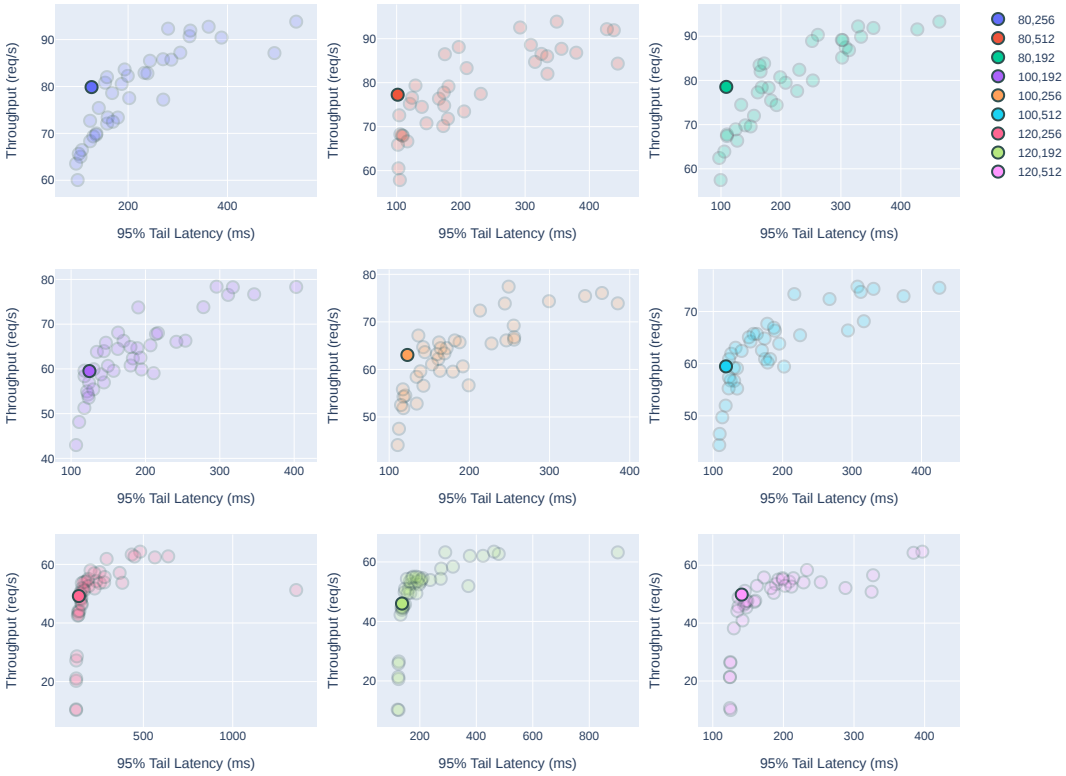
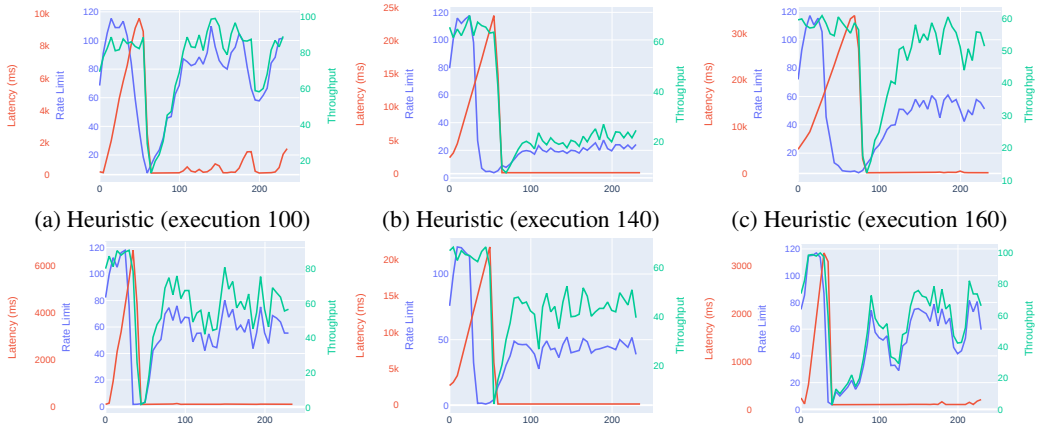


Figure 2: Visualization of PolicyShedder against the heuristics, where PolicyShedder is highlighted as solid selected points.

A.7 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 3: 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.