AutoSail : ML-based Automated Service Mesh Configuration Tuner

Kejing Huang, Qiubai Yu, Weituo Kong
Brown University
Providence, RI
{kejing_huang,qiubai_yu,weituo_kong}@brown.edu

ABSTRACT

With the wide adoption of microservice architecture in recent years, people seek ML-based optimization approaches to fine-tune service configurations for better performance. While these techniques provide sound results, they mainly focus on tuning the service nodes and leaving edge configurations.
This paper investigates tuning service mesh configurations with deep reinforcement learning. To validate technique effectiveness, we develop and present AutoSail, an ML-based automated service mesh configuration tuner. We conduct a precursory experiment along with preliminary evaluation results.

1 INTRODUCTION

Cloud computing space has gone through a major architectural shift over the past decade. Legacy cloud architecture consists of one or a few large monolithic services each running on a cloud virtual machine (VM) which requires constant human monitoring and babysitting. In a microservices architecture, large services are split into much smaller self-contained, single-purpose microservices and all microservices are deployed, monitored, and managed by the cloud orchestrater (automagically). However, these microservices require stable and optimized network communication to ensure the overall service is resiliency and efficient. Service mesh is created for this very purpose. A service mesh oversees and manages all the inter-service network communication and provides additional visibility and security. Istio, as the most popular service mesh, is currently being used by companies like Google, Airbnb, eBay, Spotify, IBM, etc [3].

Despite their advantage and wide adoption, microservices with service mesh create new challenges. Service mesh, e.g. Istio[3], provides customizable configurations on the overall network, as well as individual routes within the mesh. These configurations are usually being adjusted by experienced Kubernetes operators after constantly observing the application workload. Hiring a specialized operator to tune service mesh is both costly and time-consuming. To make things worse, each services mesh's configuration is unique to the application. A set of configurations tailored for one application could sabotage another application's performance. This is why most new microservice applications stick with the default configuration for simplicity and cost-saving, leaving plenty of room for improvement.

Machine learning-based approaches have proven to be effective in tuning complex systems [1, 9, 11, 24]. Notability, auto-tuning cloud microservice system configuration is also a popular scenario [6, 14, 21, 22]. In this project, we present AutoSail, an ML-based Automated Service MeshConfiguration Tuner. Consider a microservice application as a graph (often known as the service graph) where each node is a service and each edge is the communication between two services. While many existing configuration auto-tuners focus on adjusting individual node configurations (i.e. RPC, ThreadPool, MongoDB Parameters, etc. [21, 22]), AutoSail concentrates on tuning edges by tailoring network policies between every two connected service. In reality, tuning edge configuration is harder than tuning node for mainly two reasons. First, tuning edge requires larger-scale applications. Since computational resources are highly-scalable, service node configuration tuned in a miniature development environment could be adopted easily in a large-scale production environment. However, network resources are often much less scaleable, so only a large production-level application will saturate the limited network resources making configuration tuning desirable. As a result, edge tuning systems like AutoSail needs to run within a much larger-scale application than the alternative node tuning systems. Second, local maximum does not yield global maximum for edge tuning. For node configuration tuning, optimizing each service node likely yields the best performance for the overall system. Edge configuration tuning, on the other hand, is not the case due to shared network resources. Overassigning network bandwidth to a low-traffic service exhibits no harm locally but could impact other high-traffic services yielding global performance loss. Real performance feedback can only be observed from end-to-end testings leading to large, complex and multi-level configuration space.

The remainder of this paper is structured as follows: Section 2 discusses related machine learning auto-tunning work;

Section 3 clarifies the background knowledge on Deep Reinforcement Learning (DeepRL); Section 4 discusses the AutoSail system architecture; Section 5 presents the prototype system in our experiment; Last, Section 6 evaluates our preliminary experimental results, and examines the potential of the AutoSail system.

2 RELATED WORK

Edges optimization with reinforcement learning. There are multiple works on optimizing network communication using reinforcement learning technique. Microsoft [10] applies reinforcement learning to bandwidth estimation and congestion control in real-time communications. It learns an agent to control sending rate in an RTC system to adapt to the complex and dynamic network conditions and therefore improve the quality of the experience. To improve fault resilience, [18] uses reinforcement learning to fine-tune traffic rule parameters. In [27], Xu *et al.* implement a deep reinforcement learning based framework to solve the routing traffic problem (optimize throughput and delay) in networking. Zhang *et al.* [29] propose a reinforcement learning based approach (Critical Flow Rerouting-Reinforcement Learning) that adapts to changing traffic and network dynamics to select critical flows for given traffic. In [23], Sun *et al.* propose a scalable and intelligent network control framework called SINET using deep reinforcement learning to optimize routing policy based on the network information. Lin *et al.* [15] employs a distributed hierarchical control plane architecture coupled with a reinforcement learning based solution to minimize signaling delay in large software-defined networks. In [9], authors develops a two-level deep reinforcement learning system called AuTO to optimize routing traffic in data center networks. One is called the Peripheral System(PS) for collecting flow information and making decisions for short flows, and the other one is called the Central System for collecting global traffic information and making decisions for long flows. In [8], it embeds reinforcement learning modules inside each node of a switching network to optimize parameter setting for congestion control. We are, however, the first to tune service mesh, a simpler way to control data sharing between different components of an application. A service mesh, unlike other communication management systems, is a specialized infrastructure layer embedded right inside an application.

Edges optimization without machine learning techniques. Besides using machine learning techniques to optimize network routing, load balancing, and HTTP layers, they has been well studied in many papers using different approaches. In [16], the authors propose an optimization approach (asynchronous distributed algorithms) to solve a flow control problem to maximize the aggregate source utility over their transmission rates. In [19], the authors designed a congestion control system that scales gracefully with network capacity, providing high utilization, low queueing delay, dynamic stability, and fairness among users. Xu *et al.* [26] presents a link-state routing protocols PEFT which splits traffic over multiple paths with an exponential penalty on longer paths, leading to optimal traffic engineering. In [13], it proposes algorithm which adds explicit delay terms to the utility function measuring Quality of Service to solve a network utility maximization problem in a network with delay sensitive/insensitive traffic. In [20], authors use CoRBA algorithm to perform routing and bandwidth allocation for a given coflow to minimize the coflow completion Time. In [25], the paper presents an online coflow-aware optimization fraomework to minimize the coflow completion time. In [7], the authors propose pCoflow with in-network scheduling to improve the coflow completion Time. While [7, 20, 25] focus on optimizing routing and network bandwidth for big data applications, we focus on optimizing a broader range of network configurations for general class of microservices.

Parameters tuning or optimization with reinforcement learning. Apart from tuning network layers (communication configurations), reinforcement learning has been adopted in tuning parameters of different systems. In [12], the authors use reinforcement learning to tune hyper-parameters in machine learning model. In [28], Yang *et al.* presents a reinforcement learning MIRAS approach for better resource allocation policy in scientific workflow systems based on microservices. Mao *et al.* [17] proposes Pensieve, a reinforcement learning based approach, used for adaptive video streaming.

3 BACKGROUND

In this section, we first introduce the RL background. Then we talk about why our model adopts RL to tackle the problem.

Reinforcement Learning is a study of an agent learning through interactions with the environment. There are some key terms to describe an RL problem:

(1) **Environment**: a task, simulation, or Agent's world where it lives and interacts;
(2) **State**: the observations that the agent receives from the environment;
(3) **Reward**: feedback from the environment;
(4) **Policy**: mapping from agent's state to actions;

Imagine a situation in which an agent is interacting in an environment. The agent can choose from a variety of actions to produce various states and agent can decided actions based on the probabilities distributions across all possible actions. As a result of an action, the environment will provide a reward as a form of feedback. The agent's policy specifies the guideline of adopting the best action to maximize the overall

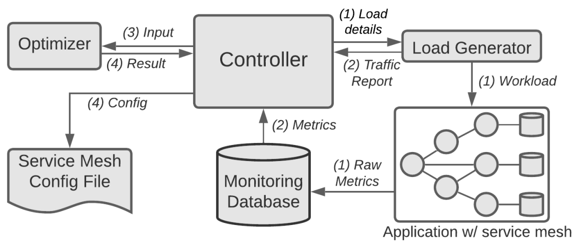

Figure 1: Illustration of AutoSail framework.

rewards. The interaction between agent and environment give rise to a sequence of states, actions, and rewards known as episode or trajectory. Each state has a value function that predicts the expected future rewards we can obtain in this state by taking the corresponding policy. In RL, we strive to learn both policy and value functions and try to maximize the expected reward when following a policy.

Policy gradient is an approach to solve RL problems. To parameterize the policy, we establish a set of parameters θ which are combinations of weights and biases in a neural network, so that it can better model and optimize the policy function. We will have solved the objective if we can find the parameters θ that maximize the reward.

The purpose of reinforcement learning is to determine the best behavior for the agent to receive the best reward. In our system, we hope that our model will help us determine the best configurations for the best performance in our system. Because RL is model-free and does not rely on accurate and solvable system models, we employ RL instead of some model-based approaches. This improves its applicability in complex networks with random and unpredictable behaviors. We use policy gradient methods (PG) to do optimization in RL. PG directly learns a function for mapping states to probability distributions over possible actions. It then samples from this distribution to pick its next move.

4 SYSTEM ARCHITECTURE

In this section, we first discuss our architecture design that aids for our experiment. We then discuss the way we abstract the service mesh configuration tuning problem as an optimization task and eventually formulate it into a reinforcement learning problem.

4.1 Overview

Figure 1 shows the architecture of the AutoSail system. AutoSail is designed to be an offline config tunning system. Our implementation workflow is mainly divided into 4 steps:

(1) Application w/ service mesh simulation;
(2) Metrics collection;
(3) Deep Reinforcement Learning;
(4) Validation on policy learning results.

The Controller is the central control system of AutoSail. It plays a crucial role in all 4 steps, and orchestrates all operations within the system. For step (1), a duplicated Application and Load Generator simulate usage and response on a production application with service mesh. AutoSail expects the test application to be deployed in a similar environment as the production system to best simulate production responses. The Controller first sends Load Details to Load Generator. Then Load Generator sends real-world-mimicking traffic Load to the Application cluster. Nest, for step (2), Controller gathers metrics from Monitoring Database and traffic report from Load Generator. It also parses both data sources into Optimizer recognizable states. Then, for step (3), these states are sent to the Optimizer as Input. Last, for step (4), Controller gathers results from the Optimizer, applies the action to the Service Mesh Config File. And the system goes back to step (1) for the next round of training.

4.2 State Space

Since the autotuner should aim to maximize the QoE(Quality of Experience) of end-users, the state space consists of two types of data - each reflecting the end-to-end performance of the application and, therefore, the QoE of end-users. First, we include the failure rates of multiple types of user requests into the states, as failure rates are among the most critical indicators of end users' QoE. Another indicator of end users' QoE would be the end-to-end response tail latency to these requests. We include both of these metrics into the states because there existed a tradeoff between failure rates and response latency. In a real-world application, unusually low latency often indicates a high failure rate, since the quickest request is one that fails early. Therefore, if we merely focus on one of them, the reinforcement learning agent will likely sacrifice the other. When evaluating each state, we desire both to play equally important roles. In conclusion, the states are formulated as

$$state = \{response\,time_1, response\,time_2, \cdots, failure\,rates\}.$$

4.3 Action Space

The agent interacts with the environment by incrementing or decreasing the tunable knobs in the service mesh. Configuration tuning problem aiming to find the optimal configuration over a continuous space is NP-Hard. Besides, countless combinations of configuration knobs make it trickier to find the optimal set of configurations. To utilize the policy gradient approach, we cut the continuous action space into discrete action space. Instead of having a neural network directly predict the value, we designed the actions to be either increment or decrement the value by a fixed amounts depending on the knobs. The increment/decrement amount is proportional to the default value of the tunable knobs. The primary reason is to reduce the effect of the networks since the randomness of networks might shadow the actual impact of the action. In addition, inspired by Wechat's overload control [30], we designed to have aggressive increments and conservative decrements. Moreover, we forced our agent to change the configurations. There's no stay still option. Considering the randomly generated initial configurations usually come with poor performances, we desired our agent to improve the performance of the service mesh better instead of playing safe.

4.4 Reward Function

The basic flow of any reinforcement learning can mainly devided into 5 steps:

(1) Observing a starting state s_t;
(2) Agent making action a_t;
(3) Observing the next state s_{t+1};
(4) Observing the reward r_t for taking action a_t;
(5) Learn from (s_t, a_t, r_t, s_{t+1}) and repeat.

We define a evaluation function for states f. If $f(s_i) > f(s_j)$, s_i is considered "better than" s_j. Since the lower failures rates and latency, the better the performance, our evaluation function for states follows a negative weighted sum format:

$$f(state_j) = \\ -(w_1 \cdot response\ time_{j,1} + \cdots + w_i \cdot failure\ rates_{j,1} \cdots)$$

The rewards should aim to measure the effect of action a_t at state s_t, we therefore designed the reward function R to be:

$$R(s_t, a_t | (s_t, a_t) \rightarrow s_{t+1}) = f(s_{t+1}) - f(s_t).$$

5 PROTOTYPE

In this section, we discuss our prototype for the experiment.

Experimental Setup. We used a Google Kubernetes Engine (GKE) cluster with eight nodes, each with dual cores, 4 GB of memory, and 100GB of disk space. This setup follows Google's recommended configuration of a GKE cluster and is representative of a real-world small-scale production system. We opted out of Google's managed Istio installation, due to their recent decision on deprecating open-source Istio support. Instead, we installed the latest open-source Istio[3] (1.9) on the GKE cluster. The AutoSail system with Kubectl (1.22), Istioctl (1.12), Prometheus (2.32), Docker (20.10), and TensorFlow (2.6) is deployed on a separate compute instance with four cores and 15GB of memory to avoid interference with the application.

Application. We employ the Sock Shop[5] application to evaluate the effectiveness of AutoSail. The Sock Shop application has 14 microservices that constitute several features of a real-world e-commerce website that sells socks. The application contains Redis, RabbitMQ, multiple instances of MongoDB, and many microservices that implement the application logic. The stock Sock Shop application does not utilize a service mesh. We created and edited application configuration files to employ Istio. The application load test consists of 62.5% GET requests, 25.0% POST requests, and 12.5% DELETE requests. 75% of the requests require a logged-in session. These divisions resemble real-world sock buyers' behavior. Because the supplied Sock Shop load generator container image is incompatible with Google's Debian image, we patch the load-test code and re-packaged the container.[1]

DRL Model. We utilized Tensorflow Keras[4] to build our reinforcement learning agent. The model contains a fully connected layer, a dropout layer, and a dense layer with softmax activation as the policy network. The model aims to minimize the loss in the form:

$$L = -\sum_{i=0}^{n} log(P_{a_i}) R_i,$$

where i represents the current step, a_i represents the action agent took at step i, and R_i the reward at step i. The trainable variables within the model are optimized with Keras' Adam optimizer with a learning rate of 0.01.

State Space. The state metrics are collected from the load generator and the Prometheus separately. After the workloads are generated, the load generator will output the failure rates of multiple user requests to the console. We used regular expression to parse the console output. Besides, we wrote a Prometheus client and a customized query to fetch the 99th percentile latency of four of the Sock Shop self-exposed methods. Finally, we concatenate all these metrics and form the state.

Action Space. The knobs of the Destination Rule [2] define policies that apply to traffic intended for a service after routing has occurred, and these knobs are related to load balancing, connection pool size from the sidecar, and outlier detection settings to detect and evict unhealthy hosts from the load balancing pool, and so on. After several tests, we discovered that fine-tuning some knobs of destination rule improves performance. As a result, we focused on the Destination Rules of the Istio configurations. In order to save training time, we identified six of congestion control related configurations as the tunable knobs for our experiment:

(1) TCP max connection: maximum number of connections to a destination host;
(2) HTTP1 max pending requests: maximum number of pending HTTP requests to a destination;
(3) HTTP max requests per connection: maximum number of requests per connection to a backend;

[1]Our patched load-test could be found in the Github repo here: https://github.com/Brown-AutoSail/load-test

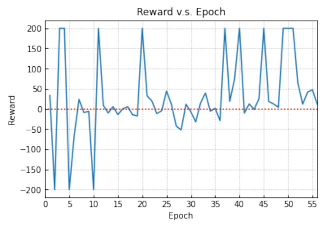

Figure 2: Rewards over epochs. Failed scenarios are included. -200 means the agent breaks the system. 200 means the agent recovers the failure. The true rewards of doing so were around 5000 and -5000, for plotting purposes, we imposed an upper bound and a lower bound to be 200 and -200.

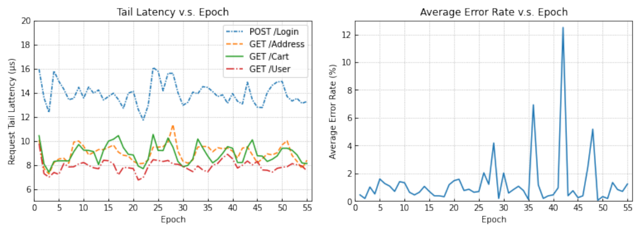

Figure 3: Final latency and failure rates over epochs. Total failure scenarios are removed from the plot, since failed scenarios are meaningless in evaluating the trend of latency and failure rates.

(4) TCP connect timeout: TCP connection timeout;
(5) HTTP2 max requests: maximum number of requests to a backend.
(6) HTTP max retries: maximum number of retries that can be outstanding to all hosts in a cluster at a given time.

The Sock Shop consists of 14 microservices. The front-end microservice was excluded since tuning the configuration of front-end is error-prone. faulty configured front-end service would ruin the entire pipeline of the learning process. Therefore, each action is to perform changes to one of the selected knobs of one of the other 13 microservices. There are in total $6 \times 2 \times 13 = 156$ actions in the action space.

Reward Function. We utilized the negative weighted sum as the evaluation function:

$$f(state_j) = \\ -(1000 \cdot p99\_latency_{j,1} + \cdots + failure\_rates_{j,1} \cdots)$$

We multiplied every latency by 1000 so that the results are at the same scale as the failure rates. To this end, latency and failure rates play equally important roles when we evaluate every single states. The reward of taking action a_i at state s_i is the difference of the evaluation of next state s_{i+1} and the evaluation of the current state s_i.

6 EVALUATION

We tested the effectiveness of policy gradient on tuning the configuration of service mesh by conducting two experiments using our prototype stated above: a) For multiple randomly generated initial state, whether the agent can improve the performance of service mesh for each of these states. b) For a specific initial state, whether the agent can consistently improve the performance of service mesh.

For the first experiment, the environment will randomly configure the service mesh. By randomly setting the initial state, we ensure that the action space is sufficiently explored and our model has sufficient knowledge of the environment. As a result, our model can "jump out" of a local maximum to find the global maximum (the potential best configurations). The number of clients and number of requests are also randomly generated. However, workload will maintain the same among epochs. Due to the time constraint, our agent was limited to make only 20 actions for each epoch. After each episode, we collected the total sum of rewards of these 20 actions without discounting, the final failure rates, and latency. We trained our agent for 55 epochs; each epoch took approximately 8~10 minutes depending on the specific load. Based on the Figure 2, we concluded three observations.

(1) From epoch 1 to 10, there exist relatively low rewards. These highly negative rewards are primarily because the agent accidentally broke the systems. The agent needs to find a way to remedy his fault. If the agent could not recover the failure, the agent would get a penalty of -5000 for making the action that broke the system. However, after the 10th epoch, the agent never broke the system again.
(2) Among all these epochs, our agent was able to get several +5000 (shown as +200) from successfully recovering the outages. In the experiment, we observed that an agent with no prior knowledge of the application (making random action) has a low chance of recovering the system. From continued training, our agent can successively recover failures on epoch 40th, 45th, 49th, 50th, and 51st.
(3) After the 30th epoch, the agent can maintain mostly positive rewards after twenty actions, indicating that the agent was able to either recover the system from failures or improve the service mesh's performance based on our evaluation function.

We also investigated the final failure rates and latency after the agent made 20 actions across epochs according to Figure 3. The failed scenarios data points are filtered and processed to maintain the trend. We primarily had two observations.

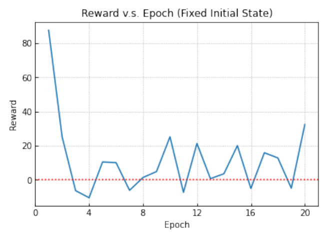

Figure 4: Rewards over epochs for fixed initial state.

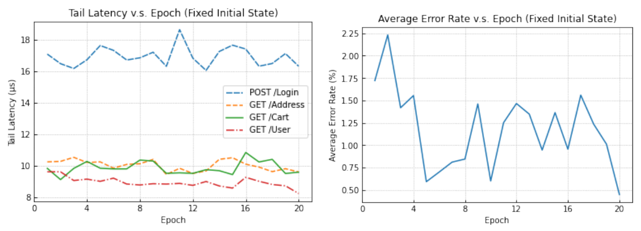

Figure 5: Final latency (left) and failure rates (right) over epochs for fixed initial state

(1) The relationship between failure rates and latency is not simply inverse proportional. For example, the corresponding latency was not minimal for the relatively large average failure rate at epoch 42. But for most cases, the tradeoff relationship exists.
(2) There are no specific values these metrics converge to. One possible reason is that 20 steps are insufficient for the agent to convert any random initial state to the "optimal" state. The other reason could probably be the agent not having a make no move option. Therefore the final metrics were constantly fluctuating.

From the first experiment, we observed that our agent is bounded by the randomized initial state, so we further investigated our agent's potential by conducting the second experiment. For every epoch, the agent starts with a fixed initial state and continues to improve the 20-action path to optimize that state. The agent loads the saved model from the previous experiment, so we assumed that this agent possessed a certain level of intelligence initially. Among the 20 epochs, the agent was able to improve the performance of the service mesh for 14 of those according to Figure 4. We also plot the latency and error rate trends across 20 epochs. Figure 5 confirms our observed trade-off between latency and error rate from the previous experiment. However, we could also observe our reward function (which is a function of both) tries to balance the trade-off between the two. The agent gets rewarded in both a latency-optimized state (epoch #14) and a failure-optimized state (epoch #20). Therefore, without loss of generality, it's safe to conclude that the agent can consistently improve the performance of any specific state given enough training.

REFERENCES

[1] [n. d.]. Automated Tuning of the JVM with Bayesian Optimization. ([n. d.]). https://www.youtube.com/watch?v=YhNl468S8CI
[2] [n. d.]. Destination Rule. ([n. d.]). https://istio.io/latest/docs/reference/config/networking/destination-rule/
[3] [n. d.]. Istio. ([n. d.]). https://istio.io/
[4] [n. d.]. Keras. ([n. d.]). https://keras.io/about/
[5] [n. d.]. Sock shop. ([n. d.]). https://microservices-demo.github.io/
[6] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 469–482. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/alipourfard
[7] Cristian Hernandez Benet, Andreas J. Kassler, Gianni Antichi, Theophilus A. Benson, and Gergely Pongracz. 2020. Providing In-network Support to Coflow Scheduling. (2020). arXiv:cs.NI/2007.02624
[8] Justin Boyan and Michael Littman. 1999. Packet Routing in Dynamically Changing Networks: A Reinforcement Learning Approach. *Advances in Neural Information Processing Systems* 6 (10 1999).
[9] Li Chen, Justinas Lingys, Kai Chen, and Feng Liu. 2018. AuTO: Scaling Deep Reinforcement Learning for Datacenter-Scale Automatic Traffic Optimization. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 191–205. https://doi.org/10.1145/3230543.3230551
[10] Joyce Fang, Martin Ellis, Bin Li, Siyao Liu, Yasaman Hosseinkashi, Michael Revow, Albert Sadovnikov, Ziyuan Liu, Peng Cheng, Sachin Ashok, David Zhao, Ross Cutler, Yan Lu, and Johannes Gehrke. 2019. Reinforcement learning for bandwidth estimation and congestion control in real-time communications. (2019). arXiv:cs.NI/1912.02222
[11] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Elliot Karro, and D. Sculley (Eds.). 2017. *Google Vizier: A Service for Black-Box Optimization*. http://www.kdd.org/kdd2017/papers/view/google-vizier-a-service-for-black-box-optimization
[12] Hadi S. Jomaa, Josif Grabocka, and Lars Schmidt-Thieme. 2019. Hyp-RL : Hyperparameter Optimization by Reinforcement Learning. (2019). arXiv:cs.LG/1906.11527
[13] Ying Li, Antonis Papachristodoulou, Mung Chiang, and Robert Calderbank. 2011. Congestion control and its stability in networks with delay sensitive traffic. *Computer Networks* 55 (01 2011), 20–32. https://doi.org/10.1016/j.comnet.2010.07.001
[14] Zhao Lucis Li, Chieh-Jan Mike Liang, Wenjia He, Lianjie Zhu, Wenjun Dai, Jin Jiang, and Guangzhong Sun. 2018. Metis: Robustly Tuning Tail Latencies of Cloud Systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 981–992. https://www.usenix.org/conference/atc18/presentation/li-zhao

[15] Shih-Chun Lin, Ian F. Akyildiz, Pu Wang, and Min Luo. 2016. QoS-Aware Adaptive Routing in Multi-layer Hierarchical Software Defined Networks: A Reinforcement Learning Approach. In *2016 IEEE International Conference on Services Computing (SCC)*. 25–33. https://doi.org/10.1109/SCC.2016.12

[16] S.H. Low and D.E. Lapsley. 1999. Optimization flow control. I. Basic algorithm and convergence. *IEEE/ACM Transactions on Networking* 7, 6 (1999), 861–874. https://doi.org/10.1109/90.811451

[17] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. 2017. Neural Adaptive Video Streaming with Pensieve. 197–210. https://doi.org/10.1145/3098822.3098843

[18] Fanfei Meng, Lalita Jagadeesan, and Marina Thottan. 2021. Model-based Reinforcement Learning for Service Mesh Fault Resiliency in a Web Application-level. (2021). arXiv:cs.DC/2110.13621

[19] F. Paganini, Zhikui Wang, J.C. Doyle, and S.H. Low. 2005. Congestion control for high performance, stability, and fairness in general networks. *IEEE/ACM Transactions on Networking* 13, 1 (2005), 43–56. https://doi.org/10.1109/TNET.2004.842216

[20] Li Shi, Junwei Zhang, Yang Liu, and Thomas Robertazzi. 2018. Coflow Scheduling in Data Centers: Routing and Bandwidth Allocation. (2018). arXiv:cs.NI/1812.06898

[21] Gagan Somashekar and Anshul Gandhi. 2021. Towards Optimal Configuration of Microservices. In *Proceedings of the 1st Workshop on Machine Learning and Systems (EuroMLSys '21)*. Association for Computing Machinery, New York, NY, USA, 7–14. https://doi.org/10.1145/3437984.3458828

[22] Akshitha Sriraman and Thomas F. Wenisch. 2018. µTune: Auto-Tuned Threading for OLDI Microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 177–194. http://www.usenix.org/conference/osdi18/presentation/sriraman

[23] Penghao Sun, Junfei Li, Zehua Guo, Yang Xu, Julong Lan, and Yuxiang Hu. 2019. SINET: Enabling Scalable Network Routing with Deep Reinforcement Learning on Partial Nodes. *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos* (2019).

[24] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-Scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1009–1024. https://doi.org/10.1145/3035918.3064029

[25] Zhaoxi Wu. 2021. Joint Coflow Optimization for Data Center Networks. *IEEE Access* 9 (2021), 108402–108410. https://doi.org/10.1109/ACCESS.2021.3102067

[26] Dahai Xu, Mung Chiang, and Jennifer Rexford. 2011. Link-State Routing With Hop-by-Hop Forwarding Can Achieve Optimal Traffic Engineering. *IEEE/ACM Transactions on Networking* 19, 6 (2011), 1717–1730. https://doi.org/10.1109/TNET.2011.2134866

[27] Zhiyuan Xu, Jian Tang, Jingsong Meng, Weiyi Zhang, Yanzhi Wang, Chi Harold Liu, and Dejun Yang. 2018. Experience-driven Networking: A Deep Reinforcement Learning based Approach. (2018). arXiv:cs.NI/1801.05757

[28] Zhe Yang, Phuong Nguyen, Haiming Jin, and Klara Nahrstedt. 2019. MIRAS: Model-based Reinforcement Learning for Microservice Resource Allocation over Scientific Workflows. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 122–132. https://doi.org/10.1109/ICDCS.2019.00021

[29] Junjie Zhang, Minghao Ye, Zehua Guo, Chen-Yu Yen, and H. Jonathan Chao. 2020. CFR-RL: Traffic Engineering With Reinforcement Learning in SDN. *IEEE Journal on Selected Areas in Communications* 38, 10 (2020), 2249–2259. https://doi.org/10.1109/JSAC.2020.3000371

[30] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. 2018. Overload Control for Scaling WeChat Microservices. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 149–161. https://doi.org/10.1145/3267809.3267823