Model-based reinforcement learning for service mesh fault resiliency in a web application-level

Fanfei Meng*
fanfeimeng2023@u.northwestern.edu
Northwestern University
USA

Lalita Jagadeesan
lalita.jagadeesan@nokia-bell-labs.com
Nokia Bell Labs
USA

Marina Thottan†
mthottan@amazon.com
Amazon Web Services
USA

ABSTRACT

Microservice-based architectures enable different aspects of applications to be created and updated independently, even after deployment. Associated technologies such as service mesh provide fault resiliency through attribute configurations that govern self-adaptive application-level behavior in response to failures, in a manner transparent to the application and constituent microservices. While this provides tremendous flexibility, the configured values of these attributes – and the relationships among them – can significantly affect the performance and fault resilience of the overall application. It is thus important to perform fault injection and load testing on the application, prior to full deployment. However, given a large number of possible attribute combinations and the complexities of the distributed system underlying microservices and service mesh architectures, it is virtually impossible to determine through traditional software development practices the worst combinations of attribute values and load settings with respect to self-adaptive application-level fault resiliency. To this end, we present a model-based reinforcement learning approach that determines the combinations of attribute and load settings that result in the most significant fault resilience behaviors at an application level. We validate our approach through a case study on a simple "request-response" service using the Istio service mesh. Our analysis shows that, even for a simple service, our model-based reinforcement learning approach outperforms a baseline selection of action parameters. Further, we show that communicative multi-agent reinforcement learning improves the performance of both the non-communicative single and multi-agent learning paradigms.

KEYWORDS

service mesh, microservices, fault resiliency, Istio, machine learning, multiple layer perceptions, model-based reinforcement learning, communicative multi-agent reinforcement learning

1 INTRODUCTION

A key trend in web application development in recent years is the advent of microservices-based architectures, in which applications are composed of small microservices that communicate with one another via distributed system mechanisms. Using open-source microservices technologies such as Kubernetes [1–3], developers can create and update different aspects of an application independently, even after deployment. At the same time, to ensure that faults in individual microservices – or delays in communication among them – do not cascade into application-level failures, microservice-based architectures increasingly include service mesh technologies such as Istio [4, 5] and Linkerd [6, 7]. These service meshes [8–10] contain associated "sidecars" [11] that monitor individual microservices for failures and delays, and perform self-adaptive actions to ensure application-level fault resilience. These actions may include, for example, bypassing problematic microservices upon consecutive errors, or ejecting them for a period of time [12–14]. The number of consecutive errors or the length of the ejection time is configured through attributes in the service mesh.

While this provides [15–20] tremendous flexibility, the configured values of these attributes – and the relationships among them – can significantly affect the performance and fault resilience of the overall application. It is thus important to perform fault injection and load testing on the application, prior to full deployment. However, given the large number of possible attribute combinations and the complexities of the distributed system underlying microservices and service mesh architectures, it is virtually impossible to determine through traditional software development practices the worst combinations of attribute values and load settings with respect to self-adaptive application-level fault resiliency.

In this paper, we present a model-based reinforcement learning approach towards service mesh fault resiliency that we call SFR2L (Service Fault Resiliency with Reinforcement Learning), which determines the combinations of attribute and load settings that result in the most significant fault resilience behaviors at an application level. Our novel contributions are as followings:

(1) To the best of our knowledge, it is the first investigation on service meshes using machine learning methods - in particular, model-based reinforcement learning - to learn the system parameters governing application-level fault resiliency.
(2) We have developed a complete model-based reinforcement learning workflow for service mesh resiliency, including data collection, service modelling, and policy learning for resiliency optimization, using a multi-faceted agent approach.
(3) We have validated our approach via a case study on the Istio service mesh using the `httpbin` "request-response" service.
(4) We provide some initial insights of the efficacy of our model-based reinforcement learning algorithms relative to certain relationships among attribute values and load settings represented in our datasets.

1.1 The Istio Service Mesh

Istio is an open-source service mesh technology for distributed and microservice architectures, that provides a transparent way to build applications. Istio's traffic management features enable service

*Work conducted while the author was at Nokia Bell Labs.
†Work conducted while the author was at Nokia Bell Labs.

monitoring and self-adaptive application-level fault resilience. In particular, Istio provides outlier detection and circuit breakers to realize fault resilience. Outlier detection enables the capacity of microservices to be limited when they are behaving anomalously, or even to be ejected for a period of time. Circuit breaking [21] is a capability that prevents microservice failures from cascading. In particular, if a microservice A calls another microservice B, which does not respond within an acceptable time period, the call can be retried or even bypassed via the circuit breaker specification.

Istio also provides fault injection [22] and load testing [23] capabilities - using the Fortio [24] load testing engine - in order to test application fault recovery. Such testing is considered critical to perform prior to application deployment to gain confidence in the fault resilience of deployed applications.

To realize these self-adaptive fault resilience mechanisms, Istio enables traffic rules to be configured for application deployment; these configurations govern the specific behaviors of outlier detection and circuit breaking. Some of the attributes of these traffic rules are depicted below, and govern the number of requests and connections allowed to a service that may be behaving anomalously, the amount of time it may be ejected and at what rate and at what detection interval, and the number of consecutive errors after which a circuit breaker will be tripped. The total threads is the number of Istio worker threads, while the total requests is the number of requests to the application - both used as configuration in Istio/Fortio load testing. Details of these attributes and configurations are at [22, 23, 25].

Traffic & Load Setting	Explanation
Max Pending Requests	Max pending req to a destination
Max Connections	Max existing connections
Max Req Per Connection	Max allowed req per connection
Ejection Time	The service ejected duration
Max Ejection	Max ejected service
Interval	Time between ejection & recovery
Consecutive Errors	Max consecutive failed requests
Total Threads	Max available threads (load)
Total Requests	Total number of requests (load)

The degree to which an application is fault resilient is heavily dependent on these attribute configurations and the relationships among them. Thus, a key challenge is to determine the most significant combination of attribute values and load settings, where the "worst" combinations of values can be used to drive load testing. However, the determination of these most significant value combinations is highly complex due to the inter-dependencies among attributes, the failure behavior of the underlying services and the communication among them, as well as the complexities of the underlying distributed system. Thus, it is impossible to determine the most significant attribute values via traditional software development practices due to sheer number of possible behaviors.

As the determination of the most significant combinations of attribute values and load settings is in essence an optimization problem, machine learning methods are well-suited to address this problem. In particular, reinforcement learning (RL) [26] is a promising approach, in which agents take actions to maximize cumulative rewards over time. In our context, the "worst" combinations can be considered as rewards, where reward computation from previous algorithmic iterations can be used to guide choices in future iterations in real-time. For Istio application fault resiliency, the "worst" behavior is when user requests to an application fail, especially under high volumes of incoming requests. The parameters below can be used as the basis for rewards (or dually, penalties).

Reward Factors	Definition
Querys Per Second (QPS)	Rate of processing incoming requests
`503` Response Rate	Failed request rate

Model-free methods in reinforcement learning. require the decision agents to take real-time actions directly on service mesh-based applications, and learn from the observed behavior. This necessitates implementation of algorithmic APIs in a service mesh (e.g. Istio) environment, likely to be very expensive and inefficient. Thus, model-based methods – in which a simulation model of the service mesh-based application is inferred through deep learning methods – would be more suitable in this context. Inspired by the ideas in [27] to infer a model for microservices resource allocation, we have developed a model-based reinforcement learning method in which "worst-case" rewards can be used to determine configuration settings that are critical in load testing for resiliency prior to application deployment. Our machine learning workflow is depicted in Figure 1, where the simulation model is inferred through multiple layer perceptions (MLP). We devise RL agents based on Q-learning, ranging from single agents to communicative multiple agents.

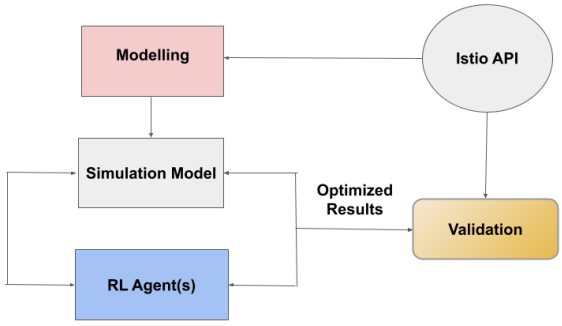

Figure 1: SFR2L Pipeline.

We validate our approach through a case study on `httpbin` – a simple "request-response" service – using the Istio service mesh. The use of `httpbin` enables us to focus primarily on the role of service mesh configuration parameters, without interference from communication delays among multiple microservices, variations in microservice topologies, and underlying distributed system issues. Our experimental results show that, even for a simple service, our model-based reinforcement learning approach outperforms a baseline selection of action parameters. Extending our analysis, we further show that communicative multi-agent reinforcement learning improves the performance of both non-communicative single and multi-agent learning paradigms. We believe our results on `httpbin` provide insight at an "atomic" level, and can be used as input into service mesh-based applications built from composite microservices.

1.2 Related Work

Microservices and their self-adaptation is an active area of research, and a comprehensive survey and taxonomy of recent work is given in [28]. However, as described in [28], there is a very limited work on application-level resiliency, as well as very little work on using machine learning in the context of service self-adaptation.

From a service resiliency perspective, two kinds of approaches have been proposed: systematic testing and formal modeling. [29] presents an infrastructure and approach for systematic testing of resiliency. However, this work does not cover the selection of the tests to be run. Our work focuses on automating such a selection through machine learning. For formal modeling, [30] presents the use of formal verification based on continuous-time Markov Chains (CTMCs) to analyze tradeoffs in service resiliency mechanisms in simple client-service interactions. [31] also uses formal verification based on CTMCs, and analyzes multiple concurrent target services as well as steady-state availability measures.

We now describe related reinforcement learning research. [27] presents a model-based reinforcement learning approach for resource allocation in scientific workflow systems based on microservices. While this paper does not address service meshes or fault resilience, our overall approach is inspired by their work. [32] proposes to utilize deep neural networks to generate Q-factors instead of storing large amounts of reward-action pairs in a hash table. Following their work, [33] presents a deterministic policy gradient algorithm to execute over continuous action spaces. For model-based reinforcement learning, [34–37] demonstrate the theoretical basis of policy gradient for model-based interactions [38–40]. With regard to multi-agent reinforcement learning (MARL), [41] introduces an efficient MARL algorithm for parallel policy optimization. [42] proposes to deploy multi-agents to optimize the traffic controls and networks, which is an important application in actual networking practice. Communication/collaboration is a common configuration in multi-agent systems [43–47] and advantageous at executing more stable, efficient and better decision-making [48–53] using decentralized Q-networks. Nevertheless, decentralized multi-agents have weak performance in the case that only small datasets are available or there are fewer state vector features for policy learning. Regarding this, [54] presents a cooperative multi-agent paradigm where the model parameters can shared by decentralized agents, while each agent preserves its own private network to make decisions. This is the groundwork for our communicative multi-service management.

Our work is, to the best of our knowledge, the first to apply model-based reinforcement learning to application-level fault resiliency for microservices and service mesh. Further, we show that communicative multi-agent reinforcement learning improves the performance of both the non-communicative single and multi-agent learning paradigm.

Our paper is structured as follows. Section 2 describes our MLP simulation model. Our model-based reinforcement learning algorithms are described in Section 3, while Section 4 presents our experimental results and discusses our conclusions and future work.

2 OUR MLP SIMULATION MODEL

We now describe our use of multiple-layer perceptrons (MLP) to infer the simulation model depicted in Figure 1.

2.1 Model Formulation

For our investigation, we focus as listed in Table 1.1 on the 7 traffic rules and 2 load settings - namely, the number of input requests to and the number of threads used by the application during load testing. We denote the feature vector representing traffic rules as $\mathbf{c}$, input requests and threads are either a_1 or a_2, which is concatenated as $\mathbf{a} = \{a_1; a_2\}$.

As described in Section 1, QPS and `503` Response Rate are two application-level responses that are essential observations in load testing. We use the following notation: N denotes the number of requests sent to the application during load testing, N_{503} the number of failed requests (`503` failures) at the end of load testing, and P_{503} the failure rate. Thus, $P_{503} = N_{503}/N$. We denote q as the application's rate of processing requests (QPS), and T as the total processing time of requests. Hence, $q = N/T$. We train a simulation model whose input-output behavior on inputs $\mathbf{a}$, $\mathbf{c}$ and outputs P_{503}, q closely simulates that of the actual Istio API under load testing. Multiple-layer perceptrons (MLP) is to simulate this relationship: $\{P_{503}; q\} = \mathbf{W}_{SM}\{\mathbf{a}; \mathbf{c}\}$, where $\mathbf{W}_{SM} = \prod_{i=1}^{l}(\mathbf{\Phi}_i + \mathbf{b}_i)$ is the well-trained l-layer MLP ($\mathbf{\Phi}_i$ is the weight of i-th layer and $\mathbf{b}_i$ is the corresponding bias).

2.2 Data Collection and Model Training

To train our MLP simulation model, we generate five structured datasets with varying parameter values as shown in Table 1, where we assume all integer values, and perform a uniform selection. (Some of the parameters have a single value, as in our experiments the impact of these parameters was negligible.)

Each data point in each dataset consists of 9 values: the 7 traffic rules c and the two load parameters a. For each data point, we run `httpbin` on the actual Istio API under Fortio load testing as described in Section 1 using configurations c and a, and record the corresponding outputs; namely, P_{503} and q. This resulting data with the configurations and corresponding responses is then provided to $\mathbf{W}_{SM}$ for training; in particular to learn the weights/bias $\mathbf{\Phi}_i$ and $\mathbf{b}_i$ for each layer. In order to test our model, we perform a 80-20 split on each dataset to obtain training and testing sets, respectively.

Table 1: Ranges of Traffic Rule, Thread, and Request Settings.

Traffic Rule & Load Settings	S1	S2	S3	S4	S5
MaxPendReq	1-7	3-7	12-18	12-18	15-30
MaxConn	1-7	3-7	1-5	10-20	5-15
Max ReqPerConn	1-7	3-7	10-16	12-18	15-30
EjecTime	3m	3m	3m	3m	3m
MaxEjec	100%	100%	4-8%	12-18%	22-30%
IntvlTime	1s	1s	1s	1s	1s
ConsecError	1	1	4-8	12-18	22-30
TotalThreads	1-5	3-7	10-16	12-18	16-20
TotalRequests	400-450	100-700	50-500	250-600	1000-2000
DatasetSize	9302	12005	20592	12310	6970

2.3 Simulation Model Evaluation

We now evaluate the performance of our MLP simulation model with respect to baseline models. [55] summarizes most common

ways of modelling networking communications, among which Logistic Regression, Linear Ridge Regression and Support Vector Regression are highlighted due to their simulation performance. We thus compare these against our 5 layer-MLP. As shown in the following table, our MLP simulation model has the best performance (mean squared error) compared to the baseline simulation models, hence we use it in our approach.

Mean Squared Error for the Models					
Simulation Model	**S1**	**S2**	**S3**	**S4**	**S5**
SVM	1.3	0.78	1.05	1.33	1.24
LogisticRegr	0.93	0.81	0.99	1.00	1.00
LinRidgeRegr	0.85	0.84	0.98	1.01	0.96
5 layer-MLP	**0.13**	**0.17**	**0.52**	**0.63**	**0.38**

3 OUR MODEL-BASED REINFORCEMENT LEARNING ALGORITHMS

We now present our model-based reinforcement algorithms. We assume load testing proceeds in m rounds; for $1 <= t <= m$, we use Round_t to denote the t^{th} round, and denote $q(t)$, $P_{503}(t)$, $a(t)$, $c(t)$ as the generalizations to rounds. We assume that all traffic rules $c(t)$ and load testing configurations $a(t)$ are (re)-set, and that all requests submitted, at the beginning of each round Round_t.

We now turn our attention to the definition of rewards. As described in Section 1, our goal is to identify the *worst* configurations with respect to fault resiliency. We thus use rewards to represent "penalties", where high rewards correspond to configurations that have a negative impact on fault resiliency. ***Configurations with high rewards can then be used as input into load testing.*** We thus define the reward $r(t)$ at Round_t as

$$r(t) = q(t) \cdot P_{503}(t). \tag{1}$$

We note that the reward $r(t)$ grows with the probability of failed requests and the rate of processing requests, hence taking into account both the probability of failure and the load on the application.

We now illustrate how to apply a single RL agent to a single service. Then we discuss the deployment of multi-agent reinforcement learning to address the complex parametric space optimization according to their collaborative relationships. Finally, multi-service resiliency optimization is illustrated using communicative decentralized learning.

3.1 Single Agent for Single Service

Firstly, we demonstrate the simplest case that only one agent and one simulation model interact with each other. In this case, only one kind of action (threads or requests) is decided by $Ag(t)$. Given a preset traffic rule $\mathbf{c}(t)$, agent $Ag(t)$ takes as input state $\mathbf{s}(t) = \{\mathbf{c}(t), a_1(t)\}$ and makes action $a_2(t)$. After that, we obtain all configurations to trigger application responses $q(t) \cdot P_{503}(t)$ to yield reward $r(t)$.

The policy of the single agent is $\pi_{\theta(t)} = a_2(t)$, and the Q-factor is $Q(\mathbf{s}, a) = E[r(t+1), r(t+2), ...|S(t) = \mathbf{s}, a_2(t) = a]$, RL model is $m(\mathbf{s}, a) = E[S(t+1)|S(t) = \mathbf{s}, a_2(t) = a]$. Our goal is to maximize the performance function $J(\theta) = E[r(1) + \alpha r(2) + \alpha^2 r(3) + ...|\pi(\theta)]$, where α is the discounted coefficient used in RL. In the implementation, the agent will search through all the actions for a given state and select the state-action pair with the highest corresponding Q-factor [56]. The policy gradient for long term

$$\nabla J(\theta) = E_\gamma[\nabla_\theta \sum_{t=0}^{t-1} log\pi(a_2(t)|\mathbf{s}(t)R(\gamma)], \tag{2}$$

where γ is the sequential action-state pair trace in time order: $\{\mathbf{s}(0), a_2(0), \mathbf{s}(1), a_2(1), ..., \mathbf{s}(t-1), a_2(t-1)\}$, $R(\gamma)$ is the reward function across the trace and $\nabla J(\theta)$ is the gradient used for network update. The single agent for single service is summarized in Algorithm 1.

Algorithm 1: Single Agent for Single Service

Input: $\mathbf{s}(1), ..., \mathbf{s}(t)$ from Round 1 to Round t

for $\mathbf{s}(t)$ **do**

  Execute the $\pi_{\theta(t)}$ to obtain the optimal action $a_2(t)$ using $\mathbf{s}(t)$;

  Combine $a(t)$ and $\mathbf{s}(t)$ to formulate input vector $\mathbf{i}(t)$ to trigger microservice response;

  Observe reward $r(t)$ to do policy gradient as per Eq. (2);

3.2 Multi-agents for Single Service

Following the previous settings, we extend the scenario into multi-agent interactions and define two kinds of collaborative relationships between two agents, respectively. Denote $a_1(t)$ as the action taken by $Ag_1(t)$, $a_2(t)$ as the action taken by $Ag_2(t)$. $\mathbf{s_1}(t)$, $\mathbf{s_2}(t)$ are corresponding state vectors, Two agents share the same reward for $\nabla J(\theta_1), \nabla J(\theta_2), \theta_1, \theta_2$ are RL Q-network parameters.

3.2.1 Independent Decision-making. In this scenario, $Ag_1(t)$ and $Ag_2(t)$ take the same state vector $\mathbf{s}(t)$ with traffic rules $\mathbf{c}(t)$ only and make actions in parallel: $\pi_{\theta_1}(t) = a_1(t)$ and $\pi_{\theta_2}(t) = a_2(t)$. After both actions are made, the input vector for microservice model is $\{\mathbf{c}(t); a_1(t); a_2(t)\}$.

3.2.2 Dependent Decision-making. Two agents are executed in order and the input for the latter agent takes into account the action of the former agent. Thus, input state vector $\mathbf{s_2}(t) = \{\mathbf{s_1}(t); a_1(t)\}$. Similarly, the input vector $\mathbf{s_1}(t) = \{\mathbf{s_2(t)}; a_2(t)\}$. All types of agent interdependencies are listed below.

Agent	State	Action (in order)
Thread Agent	7 Traffic rules + Requests	Threads
Request Agent	7 Traffic Rules + Threads	Requests
Thread&Request	7 Traffic Rules for both	Threads, Requests
Thread-Request	7 Rules-7 Rules + Threads	Threads, Requests
Request-Thread	7 Rules-7 Rules + Requests	Requests, Threads

3.3 Communicative Multi-Agents for Multi-Services

So far we explored n services that are optimized by multiple agents, but with no parameter sharing. We now introduce communication among agents for decision making, as depicted in 2. All state vectors of all agents go through the shareable Q-network SNet. SNet parameters are then input to each agent's private PNet.

Each agent calculates the rewards for their respective service, which is used to update the agent's own PNet. Outputs from each PNet are then shared with the SNet.

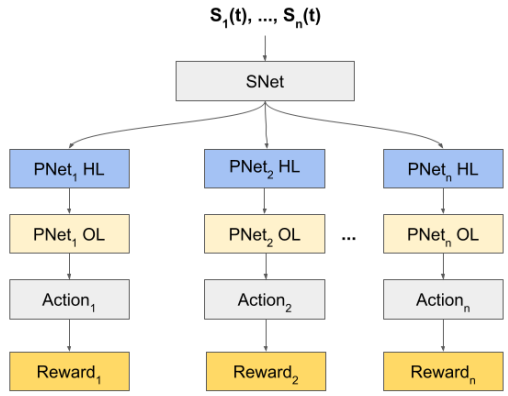

Figure 2: The configuration of communicative multi-agents.

Algorithm 2: Communicative Multi-agents

Input: $\mathbf{c_1}(1), ..., \mathbf{c_n}(1)..., \mathbf{c_1}(t), ..., \mathbf{c_n}(t)$ from Round 1 to Round t for n services

for *Round* 1 *to* t*:* **do**
  for *all agents* **do**
    Generate $\mathbf{s}_n(t)$;
    $\mathbf{s}_n(t)$ goes through SNet and obtain $\mathbf{ms_n}(t)$;
  for *all* $\mathbf{ms}_n(t)$ **do**
    $\mathbf{ms}_n(t)$ goes through respective PNet_n.
    $a_n(t) \leftarrow \pi_{\theta_s,\theta_{pn}(t)}$;
    Obtain corresponding service response ;
    Update PNet using corresponding reward $r_n(t)$ as Equation (4);
  Update SNet using all $r_n(t)$ as Equation (6);

We define the sharable network SNet with input state $S_1(t), ...S_n(t)$ and weight θ_s. The decentralized network (private) PNet with hidden and output layers and their weights are θ_{pi} where $1 <= i <= n$ is the number of agents. For the purpose of optimizing the worst-case resiliency, the reward is defined as

$$r_n(t) = q_n(t) \cdot P_{503}(t) + \beta \cdot \frac{\sum_{i=0}^{n} q_n(t) \cdot P_{503}(t)}{n}, \tag{3}$$

where β is the coefficient. After rewards are generated for each one, the respective PNet will be updated by corresponding rewards and Q-factor pair, SNet will be updated by all pairs from all service agents. As a consequence, the policy of each agent is relevant to θ_{pn} and $\pi_{\theta_s,\theta_{pn}(t)} = a_n(t)$. The long-term policy gradient for PNet_n

$$\nabla J(\theta_{pn}) = E_{\gamma_n}[\nabla_{\theta_{pn}} \sum_{t=0}^{t-1} log\pi(a_n(t)|\mathbf{ms_n}(t)R(\gamma_n)], \tag{4}$$

where $\mathbf{ms_n}(t)$ is the output vector of SNet and the input of PNet. The prediction function $\hat{f}_{\theta_s}$ of SNet is represented by

$$\mathbf{s}(\hat{t+1}) = \hat{f}_{\theta_s}(\mathbf{s_n}(t), \mathbf{ms_n}(t)). \tag{5}$$

Updating θ_s is to find the MSE minimizer of predicted $\mathbf{s}(\hat{t+1})$ and $\mathbf{s}(t+1)$

$$\theta_s = \underset{\theta_s}{\mathbf{argmin}} \frac{\sum_{\mathbf{D}} \left\| \mathbf{s_n}(t+1) - \hat{f}_{\theta_s}(\mathbf{s}(t), \mathbf{ms_n}(t)) \right\|^2}{|\mathbf{D}|}, \tag{6}$$

where training data $\mathbf{s_n}(t), \mathbf{ms_n}(t), \mathbf{s_n}(t+1) \in \mathbf{D}$ for all agents. If multiple actions are decided by agents, only agents of similar action types communicate over all services (i.e. request agents only communicate with other request agents, thread agents only communicate with other agent agents etc.). The learning paradigm for multi-services is summarized in Algorithm 2.

4 CASE STUDIES

We now describe the results of our experiments using reinforcement learning agents that implement the algorithms presented in Section 3.

Our experiments proceed in rounds: at the beginning of each round, the values for each traffic rule configuration are selected. We again use the parameter ranges in Table 1, with integer values for the traffic rules chosen uniformly from the given ranges (or using the single fixed value as given). (E.g. For S1, the max pending request is uniformly drawn from $U(1, 7)$). This gives us our $c(t)$ input for Round_t. We now use our RL algorithms to select the choices of load testing parameters for the number of requests and threads, giving us values for $a(t)$.

To test our approach, we input $c(t)$ and $a(t)$ into our MLP simulation model in each round Round_t; the output is P_{503} and $q(t)$. We repeat this for m rounds; the cumulative rewards over the m rounds is then $R_{rl} = \sum_{t=0}^{m} r(t)$. To evaluate the performance of our approach, we wish to compare the results of our RL agents against a baseline model. As no other baseline models exists to the best of our knowledge, we use a baseline model of random selection, where we use the same $c(t)$, but randomly select $a(t)$ (without the use of RL). The cumulative rewards generated by our MLP model with these baseline inputs at each round is defined as $R_{bl} = \sum_{t=0}^{m} r_{bl}(t)$. The reward ratio $Ratio_{sim} = R_{rl}/R_{bl}$, which measures the performance of our RL methods using the simulation model w.r.t. baseline selection of the $a(t)$, is depicted in Table 2 in the columns labeled "Sim."

We then also wish to compare the result of our MLP simulation model against the actual behavior of the Istio API. To this end, for each dataset used above, we record the inputs $c(t)$ and $a(t)$ (chosen by our RL agents, or by random selection for the baseline) at each Round_t. We then input these parameter values into the actual Istio API, and compute the actual rewards at each round. We thus obtain a cumulative reward on the actual Istio API using our RL agents, as well as a cumulative reward on the actual Istio API using the random baseline selection for the load testing parameters. The reward ratio $Ratio_{val}$, which measures the performance of our RL methods validated on the actual Istio API w.r.t. baseline selection of the $a(t)$, is depicted in Table 2 in the columns labeled "Val.". Our experiments use 500 epochs, where each epoch contains $m = 1000$ rounds.

Table 2: Policy Evaluations. We note that *5 means 5 services are aggregated and communicative.

Configurations \ Datasets		S1		S2		S3		S4		S5	
		Sim.	Val.	Sim.	Val.	Sim.	Val.	Sim.	Val.	Sim.	Val.
Single for Single	Request	1.03	1.01	2.71	1.77	1.75	1.31	2.05	1.81	1.31	1.29
	Thread	2.21	1.63	1.04	0.99	1.18	0.98	1.16	1.00	1.02	0.93
Multi for Single	Thread&Request	2.26	2.15	3.45	2.80	1.84	1.44	2.07	2.65	1.31	1.45
	Thread-Request	2.24	2.36	3.39	2.57	1.96	1.32	2.14	3.07	1.32	1.20
	Request-Thread	2.22	2.11	3.44	3.00	1.79	1.62	2.11	2.52	1.33	1.33
Multi for Multi	Request*5	1.01	1.00	2.96	2.30	1.77	1.43	2.05	2.87	1.33	1.30
	Thread*5	2.23	1.83	1.15	1.28	1.13	1.06	1.18	1.03	1.01	1.16
	Thread&Request*5	2.26	2.35	4.12	2.92	1.84	1.29	2.11	2.83	1.32	2.05
	Thread-Request*5	2.22	1.51	3.53	2.52	1.94	2.21	2.09	3.18	1.34	1.44
	Request-Thread*5	2.28	2.19	3.50	2.33	1.99	2.40	2.11	2.16	1.33	1.44

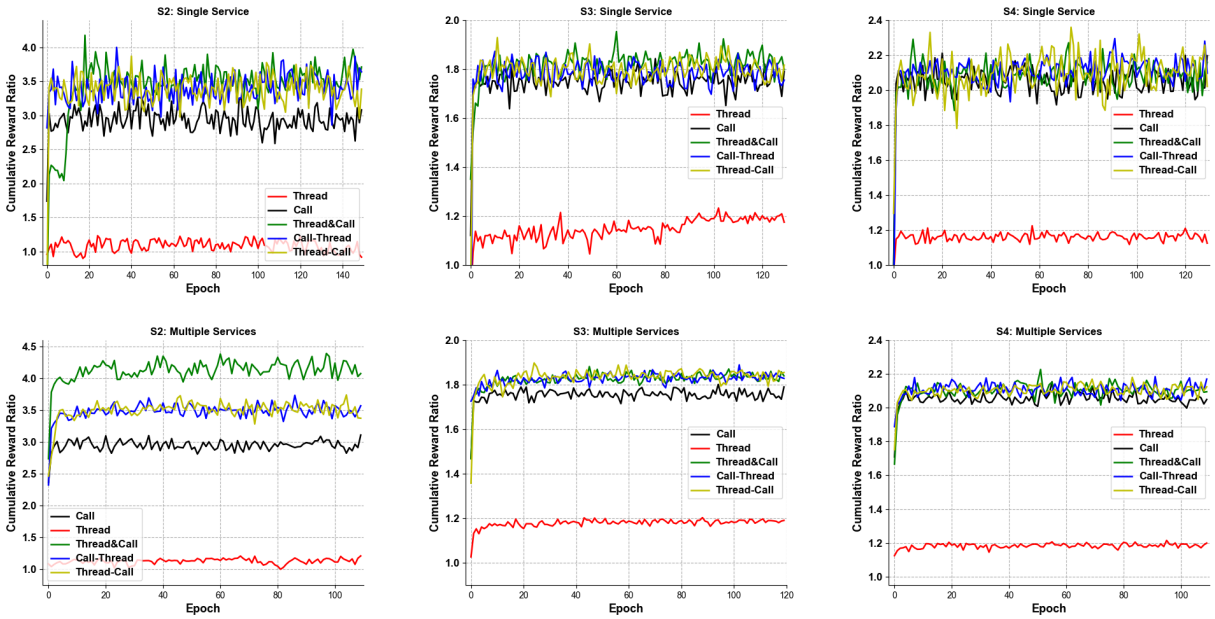

Figure 3: Cumulative reward (per epoch) ratio. Upper - single service case. Bottom - aggregated multi-services. Note that we use the term "call" interchangeably with "request".

We now examine the results in Table 2. We first observe that most of the multi-agents have higher performance than the single agent decisions, as evidenced by the higher value (recall that the value is the ratio of cumulative rewards from the RL decision w.r.t. baseline selection of load testing parameters). For instance in dataset S2, Thread&Request agents gain 27% higher rewards than Request only (3.45 to 2.71) agent in simulation and 69% higher rewards (2.80 to 1.77) in validation. Furthermore, ***multi-agents usually have higher validation accuracy than single agent***, as evidenced by the simulation value being more close to the validation value for a given data set. For example for dataset S1, the Thread only agent has 2.21 reward ratio in simulation and 1.63 in validation (36% higher), but Thread&Request agent has a 2.26 reward ratio and more accurate 2.15 validated ratio (5% higher). In addition, as shown in Figure 3, multi-services have more stable learning trends. (Note that we use the term "call" interchangeably with "request" in the figure.)

In this paper, we have comprehensively investigated how model-based reinforcement learning can aid in fault resiliency for service mesh-based applications. In particular, the configuration settings that yield the “worst-case” rewards give insight into which combinations of Istio configurations should be tested rigorously during load testing to ensure robust fault recovery. The stability of our learning trends lends confidence that the identified configurations are likely to significantly compromise application-level fault resiliency. Our experiments on a simple ”request-response” service are not subjected to interference from potential delays in microservice communication, microservices topologies, or underlying distributed systems issues. We thus view that our results can be used ”atomically” in future extensions of our approach to applications built from composite microservices.

REFERENCES

[1] Kubernetes. https://kubernetes.io.

[2] Ozair Sheikh, Serjik Dikaleh, Dharmesh Mistry, Darren Pape, and Chris Felix. Modernize digital applications with microservices management using the istio service mesh. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, pages 359–360, 2018.

[3] Matt Klein. Lyft's envoy: Experiences operating a large service mesh. 2017.

[4] Istio. https://istio.io.

[5] Lee Calcote and Zack Butcher. *Istio: Up and Running: Using a Service Mesh to Connect, Secure, Control, and Observe*. O'Reilly Media, 2019.

[6] Linkerd: Homepage. https://linkerd.io.

[7] Anjali Khatri and Vikram Khatri. *Mastering Service Mesh: Enhance, secure, and observe cloud-native applications with Istio, Linkerd, and Consul*. Packt Publishing Ltd, 2020.

[8] Kasun Indrasiri and Prabath Siriwardena. Service mesh. In *Microservices for the Enterprise*, pages 263–292. Springer, 2018.

[9] Wubin Li, Yves Lemieux, Jing Gao, Zhuofeng Zhao, and Yanbo Han. Service mesh: Challenges, state of the art, and future research opportunities. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 122–1225. IEEE, 2019.

[10] Lee Calcote. *The enterprise path to service mesh architectures*. O'Reilly Media, Incorporated, 2020.

[11] A sidecar for your service mesh. https://www.abhishek-tiwari.com/a-sidecar-for-your-service-mesh/.

[12] Haan Johng, Anup K Kalia, Jin Xiao, Maja Vuković, and Lawrence Chung. Harmonia: A continuous service monitoring framework using devops and service mesh in a complementary manner. In *International Conference on Service-Oriented Computing*, pages 151–168. Springer, 2019.

[13] XIE Xiaojing and Shyam S Govardhan. A service mesh-based load balancing and task scheduling system for deep learning applications. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 843–849. IEEE, 2020.

[14] Xing Li, Xiao Wang, and Yan Chen. Meshscope: a bottom-up approach for configuration inspection in service mesh. 2020.

[15] Branden Ghena Fanfei Meng. Research on text recognition methods based on artificial intelligence and machine learning. preprint under review, 2023.

[16] Fanfei Meng and David Demeter. Sentiment analysis with adaptive multi-head attention in transformer, 2023.

[17] Manijeh Razeghi, Arash Dehzangi, Donghai Wu, Ryan McClintock, Yiyun Zhang, Quentin Durlin, Jiakai Li, and Fanfei Meng. Antimonite-based gap-engineered type-ii superlattice materials grown by mbe and mocvd for the third generation of infrared imagers. In *Infrared Technology and Applications XLV*, volume 11002, pages 108–125. SPIE, 2019.

[18] Fanfei Meng, Lele Zhang, and Yu Chen. Fedemb: An efficient vertical and hybrid federated learning algorithm using partial network embedding.

[19] Fanfei Meng, Lele Zhang, and Yu Chen. Sample-based dynamic hierarchical trans-former with layer and head flexibility via contextual bandit.

[20] Fanfei Meng and Chen-Ao Wang. Adynamic interactive learning interface for computer science education: Program-ming decomposition tool.

[21] Circuit Breakers and Microservices Architectures. https://techblog.constantcontact.com/software-development/circuit-breakers-and-microservices/.

[22] Fault Injection. https://istio.io/latest/docs/tasks/traffic-management/fault-injection/.

[23] Circuit Breaking. https://istio.io/latest/docs/tasks/traffic-management/circuit-breaking/.

[24] Fortio. https://fortio.org/.

[25] Destination Rule. https://istio.io/latest/docs/reference/config/networking/destination-rule/.

[26] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[27] Zhe Yang, Phuong Nguyen, Haiming Jin, and Klara Nahrstedt. Miras: Model-based reinforcement learning for microservice resource allocation over scientific workflows. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 122–132, 2019.

[28] Messias Filho, Eliaquim Pimentel, Wellington Pereira, Paulo Henrique M. Maia, and Mariela I. Cortés. Self-adaptive microservice-based systems - landscape and research opportunities. In *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 167–178, 2021.

[29] Victor Heorhiadi, Shriram Rajagopalan, Hani Jamjoom, Michael K. Reiter, and Vyas Sekar. Gremlin: Systematic resilience testing of microservices. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 57–66, 2016.

[30] Nabor Mendonca, Carlos Mendes Aderaldo, Javier Cámara, and David Garlan. Model-based analysis of microservice resiliency patterns. In *2020 IEEE International Conference on Software Architecture (ICSA)*, pages 114–124. IEEE, 2020.

[31] Lalita Jategaonkar Jagadeesan and Veena B. Mendiratta. When failure is (not) an option: Reliability models for microservices architectures. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops*. IEEE, 2020.

[32] Niranjan Balachandar, Justin Dieter, and Govardana Sachithanandam Ramachandran. Collaboration of AI agents via cooperative multi-agent deep reinforcement learning. *CoRR*, abs/1907.00327, 2019.

[33] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2019.

[34] Yuping Luo, Huazhe Xu, Yuanzhi Li, Yuandong Tian, Trevor Darrell, and Tengyu Ma. Algorithmic framework for model-based deep reinforcement learning with theoretical guarantees, 2021.

[35] Anusha Nagabandi, Gregory Kahn, Ronald S. Fearing, and Sergey Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7559–7566, 2018.

[36] Jessica B. Hamrick, Abram L. Friesen, Feryal Behbahani, Arthur Guez, Fabio Viola, Sims Witherspoon, Thomas Anthony, Lars Buesing, Petar Veličković, and Théophane Weber. On the role of planning in model-based deep reinforcement learning, 2021.

[37] Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, Afroz Mohiuddin, Ryan Sepassi, George Tucker, and Henryk Michalewski. Model-based reinforcement learning for atari, 2020.

[38] Kenji Doya, Kazuyuki Samejima, Ken-ichi Katagiri, and Mitsuo Kawato. Multiple model-based reinforcement learning. *Neural computation*, 14(6):1347–1369, 2002.

[39] Bradley B Doll, Dylan A Simon, and Nathaniel D Daw. The ubiquity of model-based reinforcement learning. *Current opinion in neurobiology*, 22(6):1075–1081, 2012.

[40] Athanasios S Polydoros and Lazaros Nalpantidis. Survey of model-based reinforcement learning: Applications on robotics. *Journal of Intelligent & Robotic Systems*, 86(2):153–173, 2017.

[41] Gregory Palmer, Karl Tuyls, Daan Bloembergen, and Rahul Savani. Lenient multi-agent deep reinforcement learning, 2018.

[42] Tianshu Chu, Jie Wang, Lara Codecà, and Zhaojian Li. Multi-agent deep reinforcement learning for large-scale traffic signal control. *IEEE Transactions on Intelligent Transportation Systems*, 21(3):1086–1095, 2020.

[43] Jiechuan Jiang and Zongqing Lu. Learning attentional communication for multi-agent cooperation. *arXiv preprint arXiv:1805.07733*, 2018.

[44] Kamran Zia, Nauman Javed, Muhammad Nadeem Sial, Sohail Ahmed, Asad Amir Pirzada, and Farrukh Pervez. A distributed multi-agent rl-based autonomous spectrum allocation scheme in d2d enabled multi-tier hetnets. *IEEE Access*, 7:6733–6745, 2019.

[45] Daewoo Kim, Sangwoo Moon, David Hostallero, Wan Ju Kang, Taeyoung Lee, Kyunghwan Son, and Yung Yi. Learning to schedule communication in multi-agent reinforcement learning. *arXiv preprint arXiv:1902.01554*, 2019.

[46] Jakob Foerster, Nantas Nardelli, Gregory Farquhar, Triantafyllos Afouras, Philip HS Torr, Pushmeet Kohli, and Shimon Whiteson. Stabilising experience replay for deep multi-agent reinforcement learning. In *International conference on machine learning*, pages 1146–1155. PMLR, 2017.

[47] Tom Eccles, Yoram Bachrach, Guy Lever, Angeliki Lazaridou, and Thore Graepel. Biases for emergent communication in multi-agent reinforcement learning. *arXiv preprint arXiv:1912.05676*, 2019.

[48] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. Decentralized multi-agent reinforcement learning with networked agents: Recent advances. *arXiv preprint arXiv:1912.03821*, 2019.

[49] Hangyu Mao, Zhibo Gong, Yan Ni, and Zhen Xiao. Accnet: Actor-coordinator-critic net for" learning-to-communicate" with deep multi-agent reinforcement learning. *arXiv preprint arXiv:1706.03235*, 2017.

[50] Kaiqing Zhang, Zhuoran Yang, Han Liu, Tong Zhang, and Tamer Basar. Fully decentralized multi-agent reinforcement learning with networked agents. In *International Conference on Machine Learning*, pages 5872–5881. PMLR, 2018.

[51] Hengyuan Hu and Jakob N Foerster. Simplified action decoder for deep multi-agent reinforcement learning. *arXiv preprint arXiv:1912.02288*, 2019.

[52] Saurabh Kumar, Pararth Shah, Dilek Hakkani-Tur, and Larry Heck. Federated control with hierarchical multi-agent deep reinforcement learning. *arXiv preprint arXiv:1712.08266*, 2017.

[53] Philip Paquette, Yuchen Lu, Steven Bocco, Max O Smith, Satya Ortiz-Gagné, Jonathan K Kummerfeld, Satinder Singh, Joelle Pineau, and Aaron Courville. No press diplomacy: modeling multi-agent gameplay. *arXiv preprint arXiv:1909.02128*, 2019.

[54] Jayesh K. Gupta, M. Egorov, and Mykel J. Kochenderfer. Cooperative multi-agent control using deep reinforcement learning. In *AAMAS Workshops*, 2017.

[55] Raouf Boutaba, Mohammad A. Salahuddin, Noura Limam, Sara Ayoubi, Nashid Shahriar, Felipe Estrada Solano, and Oscar Mauricio Caicedo Rendon. A comprehensive survey on machine learning for networking: evolution, applications and research opportunities. *J. Internet Serv. Appl.*, 9(1):16:1–16:99, 2018.

[56] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.