Reinforcement Learning-based Service Assurance of Microservice Systems

Xiaojian Liu[1], Yangyang Zhang[2], Wen Gu[1], Qiao Duan[1] and Qingqing Ji[3]

[1] *Beijing University of Technology, Beijing, China*
[2] *China Electronics Standardization Institute, Beijing, China*
[3] *Chinese Academy of Sciences, Beijing, China*

Abstract

As microservices architecture has steadily emerged as the prevailing direction in software system design, the assurance of services within microservices systems has garnered increasing attention. The concept of intelligent service assurance within microservices systems offers a novel approach to addressing adaptation challenges in complex, risk-laden environments. This paper introduces a groundbreaking approach known as the Reinforcement Learning (RL) Based Service Assurance Method for Microservice Systems (RL-SAMS), which incorporates the fundamental RL principle of "improving performance through experience" into service assurance activities. Through the implementation of an intelligent service degradation mechanism, the continuity of services is ensured. Within the framework of our designed microservices system, two essential components are introduced: the Adapter Component (AC) and the RL Decision-making Component (RLDC). Each microservice is treated as an independent RL agent, resulting in the construction of a multi-agent RL decision-making architecture that balances "centralized learning and decentralized decision-making." This intelligent decision-making model undergoes training and learning, accumulating positive experiences through continuous trial and error. Experimental cases demonstrate that RL-SAMS outperforms the widely adopted *Hystrix* across various service risk scenarios, particularly excelling in intelligently critical service assurance.

Keywords

Reinforcement learning; Microservice system; Intelligent service assurance

1. Introduction

In 2014, Martin Fowler formally introduced the concept of "Microservices" through his blog post titled "Microservices." This innovative approach to software architecture involves breaking down a software system into numerous small services, each operating independently in its own process. When compared to traditional monolithic systems, microservices architectures offer several notable advantages, including the ability to deploy independently, effortless scalability, and decentralization. An increasing number of network applications have made the transition to microservices architecture, with notable examples including Amazon, Netflix, Twitter, SoundCloud, and PayPal. To give you an idea of the scale, a single page on Amazon can trigger approximately 100 to 150 microservice calls, while the Netflix system manages a staggering 5 billion microservice interactions on a daily basis [1]. It's evident that microservice architecture has progressively emerged as the predominant developmental direction for software system architecture [2][3][4].

The autonomy and collaborative interaction among microservices offer both advantages and, at the same time, present significant service reliability risks. On one hand, this autonomy entails separate operations, maintenance, and independent decision-making. This can lead to a focus on local interests at the expense of global considerations, sometimes even resulting in conflicting service assurance efforts among microservices. On the other hand, the intricate business interactions among microservices often amplify "local failures" into "cascading failures," triggering an "avalanche effect." In such cases, problem resolution becomes elusive as the root cause remains elusive.

The key to addressing these service assurance challenges lies in establishing an effective group decision-making mechanism within the microservices system. This mechanism empowers each microservice with the ability to comprehend the bigger picture and make decisions for the entire system. This paper, utilizing a reinforcement learning approach, explores a service assurance decision-making method tailored for microservices systems. Each microservice is conceptualized as an independent reinforcement learning agent. Through continuous interactions with

5th International Workshop on Experience with SQuaRE Series and its Future Direction, December 04, 2023, Seoul, Korea

liuxj@bjut.edu.cn (X. Liu);zhangyy@cesi.cn (Y. Zhang)

0000-0002-0666-4102 (X. Liu); 0009-0006-4940-8527(Y. Zhang)

the service environment and the operational and maintenance environment, the fundamental concept of "enhancing performance through experiential learning" is woven into the fabric of microservice assurance. This equips the decision-making system with the capacity to intelligently differentiate between assurance targets and to flexibly provide assurance for critical elements.

Section 2 of the paper provides a summary of related research, with a particular emphasis on the current state of research in microservice assurance technology and reinforcement learning methods. In Section 3, we present an overview of the RL-SAMS method along with an introduction to its key components. Section 4 showcases the effectiveness of the RL-SAMS method through pre-experimental results. Finally, in Section 5, we summarize the contributions of this paper and outline potential directions for future research.

2. Related Works

Technologies related to microservice assurance include service degradation technology [5][6], service fault tolerance technology [7], service elastic scaling technology [8][9], service current limiting technology [10] etc. Santos et al. [6] proposed a strategy for online service degradation based on quality of service (QoS), which aims to minimize request congestion due to lack of system resources; Combining architecture analysis method and sensitivity analysis method, Wang et al. [7] proposed a fault-tolerant strategy algorithm based on reliability criticality measurement; Coulson et al. [9] designed an automatic expansion system prototype of microservice based on supervised learning; Firmani et al. [10] put forward an API call rate limit selection strategy in order to prevent unauthorized users from achieving ultra-high SLA. Most of the existing research on microservice assurance focus on the local situation of their respective microservices. It is impossible to comprehensively consider the guarantee of service expectations from the perspective of users. One of the key problems that need to be solved is how to establish an assurance system of service for global decision-making without breaking the original distributed and independent framework of microservice.

The existing research on reinforcement learning-enabled software adaptive control can be roughly divided into: (1) Strategy generation and evolution research. Wang et al. [11] used reinforcement learning method to solve the problem of dynamic service configuration in the integrated adaptive system. Wang et al. [12] used reinforcement learning method, combined with Markov model Gaussian process, to establish a multi-agent game model, which aims to solve the problem of self-adaptive combination of services. Rao et al. [13] proposed a distributed learning mechanism to solve the problem of resource allocation in the cloud environment. Dongsun et al. [14] proposed a framework-based online planning method for self-management, which enables the software system to change and improve its plan through online RL. Amoui et al. [15] used RL in the planning process to support action selection, and clarifies why, how and when RL can benefit autonomous software systems. (2) System and environmental modelling research. Zhao et al. [16] proposed a learning framework that integrates online and offline work based on reinforcement learning and case sets. Belhaj et al. [17] put forward a framework named "autonomic container", which endows applications with run-time adaptive action capability based on RL method. With model-based reinforcement learning method, Ho HN et al. 18] used Markov process to model the environment state, which is applied for the planning and continuous optimization of adaptive software systems. Tesauro et al. [19] utilized reinforcement learning method to solve the problem of service ranking.

Regarding multi-agent RL, the representative studies in recent years include MADDPG (Multi-Agent Deep Deterministic Policy Gradient) [20] and COMA (Counterfactual Multi-Agent actor-critic) [21], both of which are based on classic Actor-Critic architecture. At present, multi-agent RL is one of the most focused and widely researched directions in reinforcement learning methods.

In summarizing the current state of research, it's clear that while various technologies and effective measures have been developed for microservice system assurance from different angles, most of them primarily address localized issues and decision-making within their own domains. As a result, they often fall short in comprehensively addressing the decision-making requirements for the overall system's assurance. The challenge now lies in merging the decision-making traits inherent to microservice architecture with the valuable insights gained from the remarkable research achievements in reinforcement learning methods within the realm of adaptive control. The objective is to empower each microservice with a global perspective and intelligent decision-making capabilities. This remains at the forefront of ongoing research efforts.

3. RL-SAMS Methodology

The comprehensive architecture of RL-SAMS is illustrated in Figure 1. Building upon the Microservice System Component (MC), we've introduced the Adapter Component (AC) and the RL Decision-making Component (RLDC). Within the MC, we've enhanced each microservice by incorporating the AC. This enhancement includes the addition of a SMM and a DCM, both of which provide interfaces for interaction with the RLDC. To keep the illustration straightforward, Figure 1 simplifies the interdependence among multiple microservices. The RLDC establishes a mechanism characterized by "centralized learning and decentralized decision-making."

The fundamental concept of "enhancing performance through experiential learning" is embedded into microservice assurance. This integration is achieved through the ongoing interactive learning of multiple agents, taking into account the effects of system operation and maintenance, user expectations, and various other state factors.

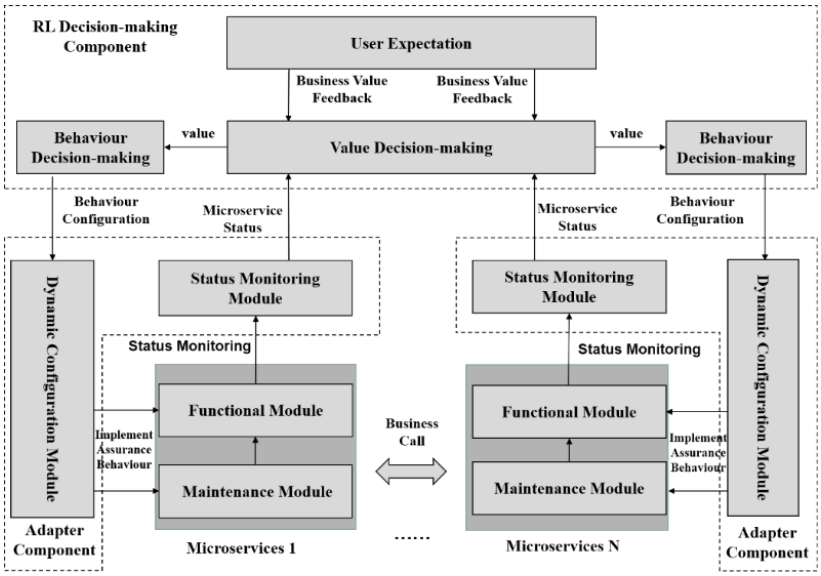

Figure 1: Architecture of RL-SAMS

3.1. Adapter Component

The core function of the AC is to provide an Interactive interface for the RLDC to perceive the running service state of the microservice system, and to timely control the configuration and implementation of various types of assurance actions. The main functional modules include a state monitoring module (SMM) and a dynamic configuration module (DCM).

1. State monitoring module (SMM). The content of state monitoring depends on the actual requirements, such as request volume, correct rate, response time, etc., and can also be specific business parameters, exception codes, etc. Spring Cloud framework provides "/metrics" endpoint, "/health" endpoint, "/trace" endpoint and other interfaces for regular microservice state monitoring. Section 4 Experiment will activate these endpoints to achieve simple state monitoring to demonstrate the effectiveness of RL-SAMS. Customized SMMs and interfaces are also suitable for the mechanism proposed in this paper.
2. Dynamic configuration module (DCM). To achieve runtime oriented dynamic assurance, it is required the RLDC have the ability to dynamically configure and execute assurance action without restarting the microservice. We establish a configuration center server to centrally manage the configuration files of each microservice, and the RLDC controls the content of each microservice configuration file according to the decision result, as well as the action of microservice configuration update, so as to realize the service assurance, as showed in Figure 2.

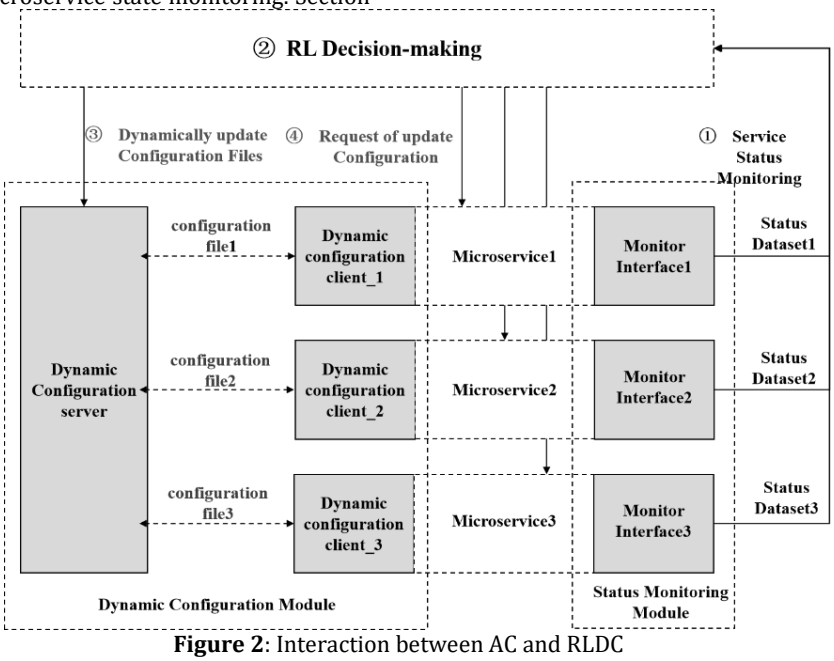

Figure 2: Interaction between AC and RLDC

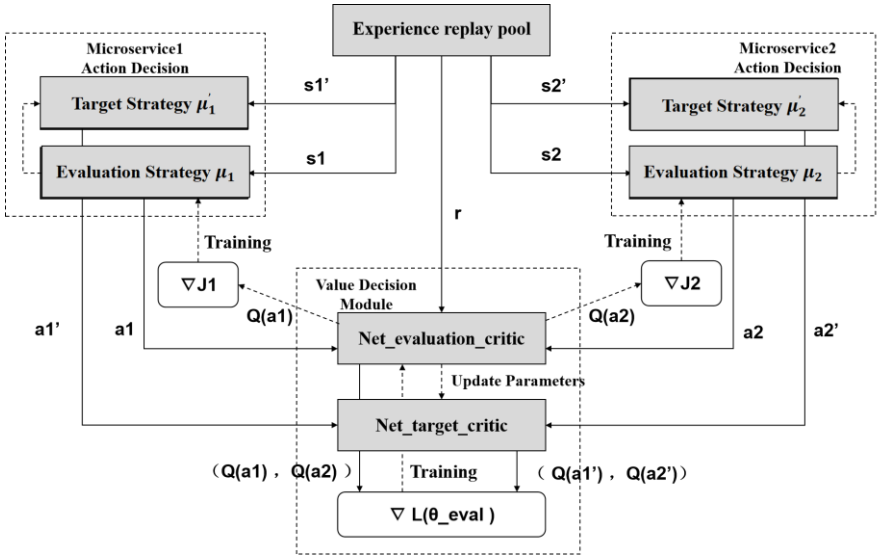

Figure 3: Framework of RLDC

3.2. RL Decision-making Component

In the RLDC, each microservice with decision-making ability is modelled as an independent agent for centralized training and decentralized execution. That is, in training stage, the learning of each agent is performed using globe states to consider strategies of other agents; in execution stage, each agent only makes decisions based on its own state perception. In addition, an experience replay pool is set up, and the experience replay mechanism is used to solve the problems of correlation between training samples and unfixed probability distribution of training samples. Each state transition are recorded as state-action pair and the corresponding reward and next state, as follows:

$$(s_1, s_2, \dots, s_n; a_1, a_2, \dots, a_n; R; s_1', s_2', \dots, s_n')$$

Where s_i is the current state of each microservice. a_i is assurance action selected by each microservice. R is reward value, such as the degree of satisfaction of various users' expectations after each assurance action is performed. s_i' is the next state of each microservice. The framework and process of the two microservices are shown in Figure 3. Each microservice corresponds to an independent "action decision" module and a shared "value decision" module. There are two strategy networks with same structure in one "action decision" module: Target strategy μ_i' and evaluation strategy μ_i, which are used to assurance decision making based on local microservice state:

1. Target strategy μ_i' takes the next state of local microservice s_i' as input, and outputs the assurance action a_i' corresponding to s_i':

$$a_i' = \mu_i'(s_i' | \theta_{target}^{\mu})$$

The target strategy μ_i' does not actively train, but periodically updates it with the parameters of the continuously learning evaluation trategy μ_i, thereby increasing the stability of the learning process. θ_{target} is the parameter of Net_target_critic.

2. Evaluation strategy μ_i takes the current state of local microservice s_i as input, and outputs the assurance action a_i corresponding to s_i:

$$a_i = \mu_i(s_i | \theta_{eval}^{\mu})$$

The evaluation strategy μ_i is continuously trained and learned based on the feedback of Q-value from "value decision" module. θ_{eval} is the parameter of $Net\_evaluation\_critic$.

Although decentralized decision-making, each microservice is closely related in business logic, so the service effect of each microservice is mostly comprehensive evaluation. Therefore, compared with MADDPG, which designs a critic module for each agent, this paper designs a shared critic module (i.e., "value decision" module) for all microservices, and outputs the corresponding *Q-value* of each microservice according to the comprehensive reward function. The "value decision" module designs two neural networks with the same structure: Value decision target network Net_target_critic and Value decision evaluation network $Net\_evaluation\_critic$, which are used to output the *Q-value* of each microservice assurance action based on the global state of the microservice system:

1. Net_target_critic takes the next state of the microservice system $(s_1', s_2', \dots, s_n')$ and the corresponding $(a_1', a_2', \dots, a_n')$ as the input, and outputs the *Q-value* corresponding to the next state of each microservice:

$$Q_1'(s_i', a_i' | \theta_{target}^{Q})$$

where θ_{target} is the parameter of Net_target_critic. Net_target_critic does not actively train and learn, but periodically updates it with the continuously learned parameters of $Net\_evaluation\_critic$ to increase the stability of the learning process.

2. $Net\_evaluation\_critic$ takes the current state of the microservice system $(s_1, s_2, \dots, s_n)$ and the corresponding $(a_1, a_2, \dots, a_n)$ as input, and outputs the *Q-value* corresponding to the current state of each microservice value:

$$Q_i(s_i, a_i | \theta_{target}^{Q})$$

where θ_{eval} is the parameter of $Net\_evaluation\_critic$. $Net\_evaluation\_critic$ periodically selects several state transition records randomly from the experience replay pool for training and learning, let's say N. The process of training and learning is the process of continuously optimizing the difference between the estimated *Q-value* and the actual *Q-value*. The loss function is defined as:

$$L(\theta_{eval}) = \frac{1}{N}\sum\left(r + \gamma * Q_i'(s_i', a_i'|\theta_{target}^{Q}) - Q_i(s_i, a_i|\theta_{eval}^{Q})\right)^2$$

where γ is the learning rate, $\gamma \in [0,1]$. The larger the γ, the more emphasis on long-term rewards in the learning process. The evaluation strategy of each microservice μ_i updates the parameters according to gradient descent (J1 and J2 in Figure 3):

$$\nabla J \approx \frac{1}{N}\sum \nabla\mu_i(s_i|\theta_{eval}^{\mu}) \cdot \nabla Q_i(s_i, a_i, \theta_{eval})$$

4. Experiments

4.1. Experimental scene

In order to verify the effectiveness of RL-SAMS, we build a user-information-querying system consisting five microservices with "VMware Workstation 16 Pro", as shown in Figure 4. The system includes three business microservices, one configuration center microservice and one registry center microservice. Each microservice is developed based on "Spring Cloud" framework[22] and deployed on an independent VMware virtual machine. The configuration of each virtual machine is as follows: memory 1GB, number of processors 1, hard disk (SCSI) 20GB, operating system Ubuntu-16.04.

Three business microservices include:

1. Two *client* microservices, $Core\_client$ and Non_core_client which are used to receive requests for querying user information, and call the $Provider\_user$ microservice to return the result to the requesting user. There is no difference in business logic between the two microservices, just to verify that the RL-SAMS has the ability to guarantee core business priority, one of the two client microservices is selected as the core business microservice.
2. One $Provider\_user$ microservice, responsible for background business processing. The microservice receives user information query requests, and returns the query results. In order to simulate the performance bottleneck of each microservice, set the $Provider\_user$ microservice to execute the information query service after sleeping for one second.

We simulate high concurrent business requests based on the performance testing framework "Locust". In the experiment, we deploy three pressure simulation modules for three business function microservices: $Core\_client$, Non_core_client, and $Provider\_user$. We set three simulation modules with different pressure cycles to simulate different pressure sources of the microservice system to verify the core business priority assurance capability of RL-SAMS in the face of different pressure sources.

4.2. Experimental Design

The experiment takes whether the two request microservices perform service degrade as action space, $a_{core} \in [\text{on}, \text{off}]$, $a_{non\_core} \in [\text{on}, \text{off}]$, and compares the average reward value of all heartbeat monitoring requests for two client microservices within 15s after each assurance action. $a_{core} = on$ means that the service degradation mechanism is enabled to ensure service continuity, and $a_{core} = off$ means the opposite. Reward function is defined as:

$$R = \frac{\sum R_{\text{CC}}}{Core\_requests} + \frac{\sum R_{\text{NC}}}{Non\_core\_requests}$$

Where,

$$R_{\text{CC}} = \begin{cases} 4, & nomal\_service \\ 1, & degrade\_service \\ -3, & service\_failure \end{cases}$$

$$R_{\text{NC}} = \begin{cases} 1, & nomal\_service \\ 0, & degrade\_service \\ -1, & service\_failure \end{cases}$$

$Core\_requests$ and $Non\_core\_requests$ are the total number of microservice state heartbeat monitoring requests sent randomly in the corresponding period, $\sum R_{\text{CC}}$ and $\sum R_{\text{NC}}$ are the sum of the heartbeat monitoring request rewards for $Core\_client$ and Non_core_client respectively. Three responses are as following:

- $normal\_service$. Returning the correct request result within the specified time;
- $degraded\_service$. The microservice is degraded and in this experiment, it is designed that a default value is returned without actually processing;
- $service\_failure$. Timing out or returning error. Different reward value is designed between $Core\_client$ and the Non_core_client to encouraging business-critical service assurance.

In RL, a 2-layer Net_target_critic and $Net\_evaluation\_critic$ are constructed based on TensorFlow. $Net\_evaluation\_critic$ updates the parameters to Net_target_critic every 200 learning. The optimization of the neural network adopts RMSprop optimizer. The learning rate γ is set to 0.9, and the exploration strategy ε is set to 0.8. The capacity of the experience replay pool is 200, and $Net\_evaluation\_critic$ randomly selects 32 sets of state transition records from the experience replay pool every 5 steps as training samples for learning, and simultaneously trains two behavioral decision evaluation strategies.

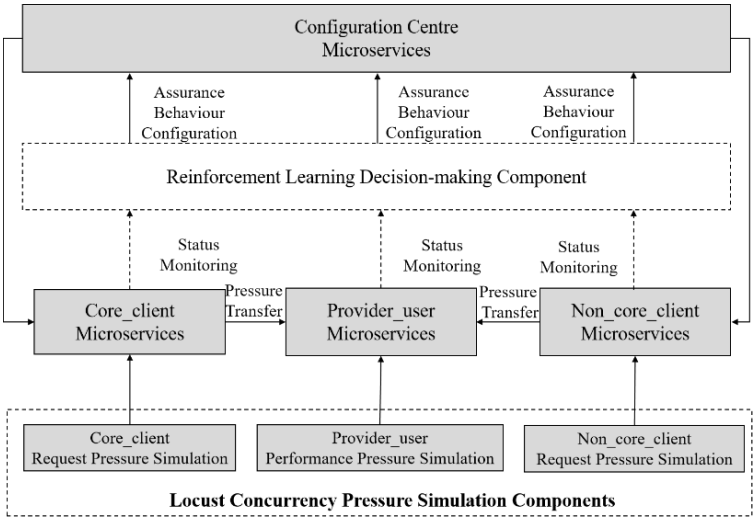

Figure 4: Experimental Scene

Table 1
Comparative Experiment Scenarios

Service risk scenarios	*Core_client* Concurrent users	*Non_core_client* Concurrent users	Expectation			
			Action	R_{CC}	R_{NC}	R
H_{JC}-H_{CC}-L_{NC}	200	50	$[a_{core} = on, a_{non\_core} = off]$	1	1	2
H_{JC}-L_{CC}-H_{NC}	50	200	$[a_{core} = off, a_{non\_core} = on]$	4	0	4
L_{JC}-L_{CC}-L_{NC}	50	50	$[a_{core} = off, a_{non\_core} = off]$	4	1	5
H_{JC}-L_{CC}-L_{NC}	100	100	$[a_{core} = off, a_{non\_core} = on]$	4	0	4
H_{JC}-H_{CC}-H_{NC}	200	200	$[a_{core} = on, a_{non\_core} = off]$	1	0	1

4.3. Comparative Experiment

4.3.1. Effectiveness Analysis

Experiment takes the widely used *Hystrix*[23] as baseline method, and compares assurance effect between the *Hytrix* service circuit breaker mechanism and RL-SAMS in five service risk scenarios shown in Table 1. In addition, the service effect without any assurance method, named "Blank" in Figure 5, will be compared as another baseline to verify the successful implementation of *Hystrix* and RL-SAMS.Table 1 shows five different service risk scenarios and expected optimal decision action and average reward. The name of service risk scenarios is combined by three fields, $X1_{JC}$–$X2_{CC}$–$X3_{NC}$, corresponding different concurrent pressure models. $X1_{JC}$ is joint concurrent field, meaning if requests from both $Core\_requests$ and $Non\_core\_requests$ together will achieve performance saturation. $X2_{CC}$ and $X3_{NC}$ is independent concurrent fields, meaning if requests from $Core\_requests$ or $Non\_core\_requests$ respectively will achieve performance saturation. H means high concurrent pressure. L means low concurrent pressure. The preliminary experiments indicate that around 150 concurrent users can subject the microservices in this experiment to high concurrency pressure.

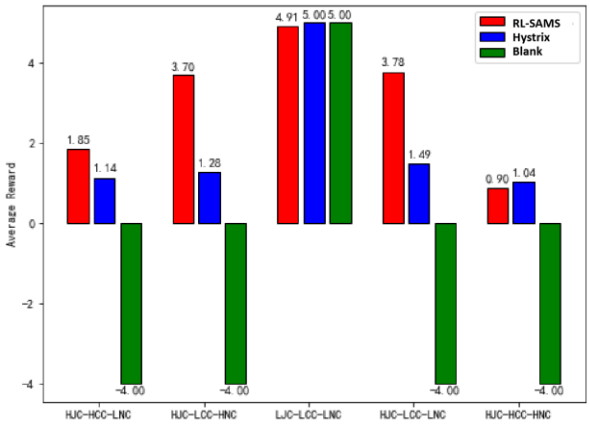

Figure 5: Comparative Experiment

The average reward value of heartbeat monitoring requests for three different service assurance methods in five service risk scenarios is shown in Figure 5.

In all H_{JC} scenarios: (1) The "Blank" method will cause the response time of all requests to time out. According to the reward function, the average reward value is -4. (2) Using the "*Hystrix*" method, whether it is high independent concurrency (H_{CC} or H_{NC}), will activate circuit breakers of both two request microservices. The average reward value is 1. Due to the existence of the retransmission mechanism in "*Hystrix*", the average reward value fluctuates in the range of 1+0.2. (3) By comparing the effects of H_{JC}–H_{CC}–L_{NC}, H_{JC}–L_{CC}–H_{NC}, H_{JC}–L_{CC}–L_{NC}, it is verified that the decision model trained by the RL-SAMS will intelligently and selectively execute the degrade of the microservices according to source of pressure. In

H_{JC}–H_{CC}–L_{NC}, since H_{CC} causes H_{JC}, *Core_requests* is impossible to assurance. So, it is best to degrade its service to assurance *Non_core_requests*; In H_{JC}–L_{CC}–H_{NC}, since H_{NC} causes H_{JC}, it is best to degrade *Non_core_requests* to assurance *Core_requests*; In H_{JC}–L_{CC}–L_{NC}, *Core_requests* and *Non_core_requests* together cause *HJ C*, it is also best to degrade and sacrifice *Non_core_requests* to assurance *Core_requests*, according to reward function.

The experiment verify that RL-SAMS can not only effectively select the assurance action, but also distinguish the degraded objects according to the source of the service risk, so as to realize intelligent elastic Microservice System assurance.

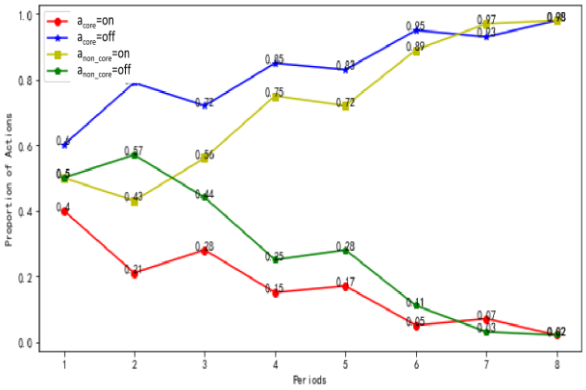

Figure 6: RL process in H_{JC}-L_{CC}-H_{NC}

4.3.2. Model Accuracy and Training Process Analysis

During the model training process, two Locust modules for handling requests as microservices continuously simulate concurrent request pressures with a random cycle duration of 1800 seconds. Considering coverage of risk scenarios for five types of services and RL state space control to shorten the learning cycle, the random range for concurrent users is set to [0, 50, 100, 150, 200]. Logs record the state of each step and the selection of safeguarding actions during the model training process. Taking service risk scenario H_{JC}-L_{CC}-H_{NC} as an example, Figure 6 presents the proportion of assurance actions at each stage of training.Due to the random nature of simulating concurrent request pressures, H_{JC}-L_{CC}-H_{NC} does not occur continuously. The number of cycles in Figure 6 refers to the extraction of all assurance action selection records when H_{JC}-L_{CC}-H_{NC} occurs throughout the entire training process. These records are sorted chronologically, and every 100 data points are used to calculate the proportion of assurance actions in a Period. The decision of whether to degrade *Core_client* and *Non_core_client* microservices to break their concurrent requests will be made. As shown in Figure 6, in Period 1, the intelligent agents of the two request microservices almost randomly decide whether to activate the degradation. Since both client microservices experience low concurrent pressure, they both exhibit a trend of not activating degradation in Period 2, resulting in an increase in the proportion of $[a_{core} = \text{off}, a_{non\_core} = \text{off}]$. Under the influence of the "value decision" module, *Core_client* and *Non_core_client* will receive the maximum reward values with $[a_{core} = \text{off}, a_{non\_core} = \text{on}]$, and their corresponding *Q-values* will also be the highest. Therefore, as training progresses, the proportion of $[a_{core} = \text{off}, a_{non\_core} = \text{on}]$ increases. After Period 6, the proportion of $[a_{core} = \text{off}, a_{non\_core} = \text{on}]$ exceeds 90% and stabilizes, reaching 98% in Period 8. In other words, in service risk scenario H_{JC}-L_{CC}-H_{NC}, RL-SAMS can, with a probability of 98% * 98% = 96%, ensure the normal service of the *Core_client* by only degrading the concurrent requests of the *Non_core_client*. The accuracy performance in other service risk scenarios is similar.

5. Conclusion

This paper introduces an innovative decision-making method for microservice systems, leveraging reinforcement learning principles. It seamlessly incorporates the core concept of "enhancing performance through experiential learning" into service assurance processes within the microservices architecture. The flexible assurance capability targeting critical assurance components paves the way for novel approaches to intelligent service assurance and maintenance. Through a thorough analysis and validation via case experiments, RL-SAMS demonstrates its prowess across various service risk scenarios, particularly excelling in its ability to intelligently differentiate key assurance elements and proactively ensure the continuity of core business operations.

While this paper has introduced reinforcement learning methods into service assurance activities within microservice systems, there are still many aspects that require further research and exploration. These include:

- Efficient Learning with Expanding State and Action Spaces: Reinforcement learning is fundamentally about accumulating experiential knowledge to maximize rewards and minimize losses. As the state and action spaces grow, the cost of model training and learning also increases rapidly. It will be necessary to investigate and improve methods for accumulating positive experiences more efficiently and enhancing convergence rates.
- Decentralized Training and Centralized Learning: The approach taken in this paper involves centralized training and learning. However, in real-world scenarios where microservices come from different providers, there may be obstacles to sharing operational data. Addressing how to limit data sharing while enabling decentralized training for individual microservices and centralized learning of experiences is a pressing challenge.
- Integration with Log Analysis and Risk Prediction: Exploring how to combine reinforcement learning with log analysis and risk prediction to leverage prior knowledge and accelerate learning efficiency is an area worth investigating. Integrating reinforcement learning with existing systems for proactive risk management and

incident response can enhance the overall effectiveness of service assurance activities.

These areas of research and improvement will contribute to the further development and refinement of reinforcement learning methods in the context of microservices and service assurance.

References

[1] Xiang zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. Poster: Benchmarking microservice systems for software engineering research. In 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), pages 323–324. IEEE, 2018.

[2] Holger Knoche and Wilhelm Hasselbring. Using microservices for legacy software modernization. IEEE Software, 35(3):44–49, 2018.

[3] Florian Rademacher, Jonas Sorgalla, and Sabine Sachweh. Challenges of domain-driven microservice design: A model-driven perspective. IEEE Software, 35(3):36–43, 2018.

[4] Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi. Cloud container technologies: a state-of-the-art review. IEEE Transactions on Cloud Computing, 7(3):677–692, 2017.

[5] Zhizhen Zhong, Jipu Li, Nan Hua, Gustavo B Figueiredo, Yanhe Li, Xiaoping Zheng, and Biswanath Mukherjee. On qos-assured degraded provisioning in service-differentiated multi-layer elastic optical networks. In 2016 IEEE Global Communications Conference (GLOBECOM), pages 1–5. IEEE, 2016.

[6] Alex S Santos, Andre K Horota, Zhizhen Zhong, Juliana De Santi, Gustavo B Figueiredo, Massimo Tornatore, and Biswanath Mukherjee. An online strategy for service degradation with proportional qos in elastic optical networks. In 2018 IEEE International Conference on Communications (ICC), pages 1–6. IEEE, 2018.

[7] Lei Wang. Architecture-based reliability-sensitive criticality measure for fault-tolerance cloud applications. IEEE Transactions on Parallel and Distributed Systems, 30(11):2408–2421, 2019.

[8] Chenhao Qu, Rodrigo N Calheiros, and Rajkumar Buyya. Auto-scaling web applications in clouds: A taxonomy and survey. ACM Computing Surveys (CSUR), 51(4):1–33, 2018.

[9] Nathan Cruz Coulson, Stelios Sotiriadis, and Nik Bessis. Adaptive microservice scaling for elastic applications. IEEE Internet of Things Journal, 7(5):4195–4202, 2020.

[10] Donatella Firmani, Francesco Leotta, and Massimo Mecella. On computing throttling rate limits in web apis through statistical inference. In 2019 IEEE International Conference on Web Services (ICWS), pages 418–425. IEEE, 2019.

[11] Hongbing Wang, Xiaojun Wang, Xingguo Hu, Xingzhi Zhang, and Mingzhu Gu. A multi-agent reinforcement learning approach to dynamic service composition. Information Sciences, 363:96–119, 2016.

[12] Hongbing Wang, Qin Wu, Xin Chen, Qi Yu, Zibin Zheng, and Athman Bouguettaya. Adaptive and dynamic service composition via multiagent reinforcement learning. In 2014 IEEE international conference on web services, pages 447–454. IEEE, 2014.

[13] Jia Rao, Xiangping Bu, Kun Wang, and Cheng-Zhong Xu. Self-adaptive provisioning of virtualized resources in cloud computing. In Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems, pages 129–130, 2011.

[14] Dongsun Kim and Sooyong Park. Reinforcement learning-based dynamic adaptation planning method for architecture-based selfmanaged software. In 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, pages 76–85. IEEE, 2009.

[15] Mehdi Amoui, Mazeiar Salehie, Siavash Mirarab, and Ladan Tahvildari. Adaptive action selection in autonomic software using reinforcement learning. In Fourth International Conference on Autonomic and Autonomous Systems (ICAS'08), pages 175–181. IEEE, 2008.

[16] Tianqi Zhao, Wei Zhang, Haiyan Zhao, and Zhi Jin. A reinforcement learning-based framework for the generation and evolution of adaptation rules. In 2017 IEEE International Conference on Autonomic Computing (ICAC), pages 103–112. IEEE, 2017.

[17] Nabila Belhaj, Djamel Belaïd, and Hamid Mukhtar. Framework for building self-adaptive component applications based on reinforcement learning. In 2018 IEEE International Conference on Services Computing (SCC), pages 17–24. IEEE, 2018.

[18] Han Nguyen Ho and Eunseok Lee. Model-based reinforcement learning approach for planning in self-adaptive software system. In Proceedings of the 9th International Conference on Ubiquitous Information Management and Communication, pages 1–8, 2015.

[19] Gerald Tesauro, Nicholas K Jong, Rajarshi Das, and Mohamed N Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. In 2006 IEEE International Conference on Autonomic Computing, pages 65–73. IEEE, 2006.

[20] Ryan Lowe, Yi I Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative competitive environments. Advances in neural information processing systems, 30, 2017.

[21] Jakob Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. Counterfactual multi-agent policy gradients. In Proceedings of the AAAI conference on artificial intelligence, volume 32, 2018.

[22] Cosmina I, Cosmina I. Spring microservices with spring cloud[J]. Pivotal certified professional spring developer exam: a study guide, 2017: 435-459.

[23] Molchanov H, Zhmaiev A. Circuit breaker in systems based on microservices architecture[J]. 2018.