

# SealOS+: A Sealos-based Approach for Adaptive Resource Optimization Under Dynamic Workloads for Securities Trading System

1<sup>st</sup> Haojie Jia  
Nanyang Normal University  
Nanyang, Henan, China  
rzkj@nynu.edu.cn

2<sup>nd</sup> Zhenhao Li  
Nanyang Normal University  
Nanyang, Henan, China  
zh.li@nynu.edu.cn

3<sup>rd</sup> Gen Li  
Nanyang Normal University  
Nanyang, Henan, China  
ligen@nynu.edu.cn

4<sup>th</sup> Minxian Xu\*  
Shenzhen Institutes of Advanced Technology  
Chinese Academy of Sciences  
Shenzhen, Guangdong, China  
mx.xu@siat.ac.cn

5<sup>th</sup> Kejiang Ye  
Shenzhen Institutes of Advanced Technology  
Chinese Academy of Sciences  
Shenzhen, Guangdong, China  
kj.ye@siat.ac.cn

**Abstract**—As securities trading systems transition to a microservices architecture, optimizing system performance presents challenges such as inefficient resource scheduling and high service response delays. Existing container orchestration platforms lack tailored performance optimization mechanisms for trading scenarios, making it difficult to meet the stringent 50ms response time requirement imposed by exchanges. This paper introduces SealOS+, a Sealos-based performance optimization approach for securities trading, incorporating an adaptive resource scheduling algorithm leveraging deep reinforcement learning, a three-level caching mechanism for trading operations, and a Long Short-Term Memory (LSTM) based load prediction model. Real-world deployment at a securities exchange demonstrates that the optimized system achieves an average CPU utilization of 78%, reduces transaction response time to 105ms, and reaches a peak processing capacity of 15,000 transactions per second, effectively meeting the rigorous performance and reliability demands of securities trading.

**Index Terms**—Securities Trading System, Resource Overhead, Performance Analysis, System Optimization, Load Monitoring

## I. INTRODUCTION

Microservices architecture has emerged as the preferred choice for large-scale distributed systems due to its superior scalability, elasticity, and agility over traditional monolithic architectures [1]. This shift is particularly advantageous for securities trading systems, which must process massive concurrent transactions with millisecond-level response time (RT). Unlike monolithic systems that struggle with performance bottlenecks—especially during peak trading periods such as market opening and closing—microservices architectures decompose trading systems into independently scalable services (e.g., order processing, matching engine, clearing, and settlement). This decomposition enables better load distribution

and resource utilization while improving fault tolerance and maintainability [2].

Despite these advantages, transitioning securities trading systems to a microservices architecture introduces unique challenges. The first major challenge is extreme latency sensitivity, as modern exchanges impose strict service-level agreements (SLAs) requiring end-to-end order processing to be completed within 50 milliseconds. Any additional delay can lead to missed trading opportunities, financial losses, and a competitive disadvantage. Second, securities trading exhibits highly dynamic workloads, characterized by tidal and bursty loads. Traditional elastic scaling approaches—such as auto-scaling groups in cloud environments—often fail to respond quickly enough to sudden trading surges, leading to resource overprovisioning or performance degradation. Third, microservices introduce significant network overhead due to frequent cross-service and cross-node communication, increasing response latency. Service mesh and API gateway solutions attempt to mitigate these issues but are not specifically optimized for ultra-low-latency trading scenarios, leading to inefficiencies.

To address these challenges, we propose a performance optimization scheme called **SealOS+** for securities trading systems based on Sealos [3], a lightweight cloud operating system that enhances Kubernetes' scheduling efficiency. Our approach integrates several key techniques to optimize performance under dynamic trading workloads: First, we design an intelligent scheduling algorithm that dynamically adjusts resource allocation based on real-time trading load characteristics. By leveraging deep reinforcement learning, the scheduler learns optimal resource allocation strategies, reducing response latency while maintaining high system throughput [4]. Second, to mitigate the overhead of frequent inter-service communication, we introduce a caching architecture comprising local memory caching, distributed caching, and persistent

\*Corresponding author

storage. This hierarchical caching strategy significantly reduces database query latency and network transmission delays, thereby enhancing overall system responsiveness [5]. Third, given the bursty nature of trading workloads, we develop a Long Short-Term Memory (LSTM)-based model [6] that accurately predicts trading volume fluctuations. By proactively adjusting resource allocation ahead of anticipated load spikes, the system achieves more efficient utilization and prevents bottlenecks. The **contributions** of this paper are:

- An adaptive resource scheduling algorithm using deep reinforcement learning to dynamically allocate computing resources in response to changing trading workloads.
- A three-level caching architecture to minimize service response time by efficiently managing frequently accessed trading data.
- Through real-world deployment at a securities exchange, our optimized system demonstrates substantial performance gains, achieving an average CPU utilization of 78%, reducing transaction response time to 105ms, and supporting peak processing capacities of 15,000 TPS.

The rest of the paper is organized as follows: Section II discusses the related work. Section III introduces the system design of our proposed approach and the system model is presented in Section IV. The extensive experiments are shown in Section V, and the conclusions are given in Section VI.

## II. RELATED WORK

Existing research on container orchestration and resource scheduling can be broadly classified into three categories: (1) scheduling and load balancing techniques, (2) container orchestration for performance optimization, and (3) machine learning-based resource management. While these studies provide valuable insights into general cloud computing environments, they do not fully address the unique challenges posed by bursty trading workloads, ultra-low-latency constraints, and real-time adaptive resource allocation.

### A. Scheduling and Load Balancing Techniques

Several studies have explored scheduling strategies to improve resource utilization and load balancing in cloud and containerized environments. Senthil et al. [7] introduced a hybrid Genetic Algorithm-Ant Colony Optimization (GA-ACO) scheme for energy-efficient load balancing, but it does not account for the unpredictable surges in trading workloads. Nguyen and Kim [8] proposed a Kubernetes-based load balancing approach to enhance scalability; however, their method does not incorporate business-specific constraints, such as trading latency requirements. Jeong and Jeong [9] developed ARAScaler, an adaptive resource scaling system, but it fails to meet the millisecond-level response time needed for financial trading systems.

Existing scheduling approaches either focus on general cloud applications or prioritize energy efficiency over real-time responsiveness. Unlike these works, our Sealos-based approach employs deep reinforcement learning (DRL) to dynamically adjust scheduling policies based on real-time trading

workload patterns, ensuring minimal response latency and efficient resource utilization.

### B. Container Orchestration for Performance Optimization

Several researchers have investigated performance optimization in containerized environments. Reddy et al. [10] analyzed Kubernetes performance across OpenStack, virtual machines, and bare-metal deployments, identifying significant performance variations but not proposing optimization strategies. Marchese and Tomarchio [11] designed a network SLO-based container orchestration framework, though its centralized architecture introduces bottlenecks during high-frequency trading peaks. Carrión [12] and Casalicchio [13] surveyed Kubernetes scheduling challenges, emphasizing the inadequacies of existing strategies in handling bursty workloads.

Prior research lacks end-to-end performance optimization strategies tailored to financial trading systems. In contrast, our work introduces a three-level caching mechanism (local memory, distributed cache, and persistent storage) to reduce inter-service communication latency, improving responsiveness under high transaction loads.

### C. Machine Learning-Based Resource Management

Recent advancements have applied machine learning to container scheduling and resource allocation. Xu and Buyya [14] proposed a Brownout-based adaptive resource management framework for cloud environments, focusing on general cloud applications rather than trading workloads. Zhong et al. [15] categorized machine learning-based container orchestration techniques, identifying high inference overhead as a major limitation. Jian et al. [16] developed DRS, a deep reinforcement learning-based Kubernetes scheduler that dynamically optimizes container placement and resource allocation. However, it primarily targets load balancing without addressing trading-specific constraints such as sub-50ms latency. Li et al. [17] proposed a reinforcement learning-based rate limiter for e-commerce services, which does not meet the high-frequency trading requirements.

Existing machine learning-based approaches either lack real-time inference capabilities or focus on general cloud applications rather than trading scenarios. Our work introduces an LSTM-based trading volume prediction model with 92% accuracy, enabling proactive resource allocation based on forecasted workload fluctuations. This predictive capability is crucial for handling the rapid and unpredictable trading surges that occur during market events.

To summarize, our Sealos-based approach bridges these gaps by integrating deep reinforcement learning for adaptive scheduling, a three-level caching strategy for optimized data access, and an LSTM-based prediction model for proactive resource allocation. Through real-world deployment, our optimized system significantly improves resource efficiency, response times, and overall trading system performance compared to existing solutions.

### III. SYSTEM ARCHITECTURE DESIGN

In this section, we first introduce the design of original Sealos, then discuss the bottleneck that should be improved when applying it to securities trading systems.

#### A. Architecture Overview

SealOS+ employs a layered architecture consisting of three primary layers: the system interface layer, application management layer, and cloud kernel driver layer. As illustrated in Fig. 1, the system interface layer provides a unified access point through APIs, command-line interfaces (CLI), and desktop interfaces, ensuring seamless interaction with the system. The application management layer, which serves as the core of the system, manages the entire application lifecycle using lightweight containers while offering essential features such as orchestration, auto-scaling, and health monitoring. Beneath it, the cloud kernel driver layer is responsible for resource management and scheduling, abstracting and standardizing network, storage, and compute resources. This layered architecture ensures modularity, scalability, and simplified maintenance, while standardized communication interfaces between layers enable loose coupling and flexible customization, making the system adaptable to diverse trading workloads.

Sealos' core components consist of the cluster manager, application controller, and resource scheduler. The cluster manager oversees node lifecycles, ensuring system stability and reliability. The application controller leverages a declarative API to manage deployments, updates, and rollbacks, ensuring applications maintain their desired states. The resource scheduler implements global resource balancing strategies to optimize workload distribution. These components communicate through a fault-tolerant, asynchronous message bus, enhancing system concurrency and resilience.

Sealos utilizes an event-driven model to ensure loose coupling among its components. User requests first pass through an API gateway for routing and then undergo authentication and authorization by the cluster manager. Deployment requests are handled by a workflow engine, which breaks down complex operations into atomic tasks for efficient execution. Resource allocation follows a two-phase commit protocol, ensuring consistency across distributed nodes. An event bus facilitates asynchronous parallel processing, while the workflow engine manages state tracking, error handling, and rollback/recovery mechanisms.

#### B. System Improvement Design

1) *Performance Bottleneck Analysis of Existing System:* Through an in-depth analysis of securities trading system run-time data as well as the system performance requirement, we identified three major performance bottlenecks when applying Sealos to trading workloads:

**High Network Communication Overhead:** Network communication contributes to 45% of the total response time, with cross-node data synchronization latency averaging 280ms. This delay significantly impacts real-time transaction processing, making it difficult to meet stringent latency requirements.

**Resource Scheduling Inefficiencies:** During the resource scheduling phase, state computation and update operations consume excessive CPU resources, causing single-node CPU utilization to peak at 85%. As a result, scheduling latency exceeds 400ms, leading to delays in workload distribution and scaling.

**Storage Bottlenecks from Metadata Operations:** Frequent metadata operations trigger excessive disk I/O, generating approximately 15,000 random write requests per second. This results in an average write latency of 30ms, severely degrading overall system performance and response times.

2) *Overall Improvement Scheme:* To address these challenges, we propose SealOS+, an enhanced Sealos-based system optimized for high-performance securities trading. Our optimizations focus on reducing network latency, improving resource scheduling, and enhancing storage efficiency to ensure real-time, low-latency operations. Based on a thorough analysis of performance bottlenecks, we developed a multi-layered optimization approach. At the network level, an intelligent routing algorithm reduced cross-node communication latency by 58%, achieving a latency of 120ms. At the computing level, an asynchronous processing mechanism and parallel task decomposition improved CPU utilization. At the storage level, an enhanced caching strategy enabled 80% of read operations to directly access memory, reducing write latency to 9ms. These optimizations resulted in a 65-second system startup time, a 99.95% deployment success rate, and a 75% reduction in node load variance. The new architecture excels in high-concurrency scenarios, handling up to 20,000 API requests per second while maintaining a 95th percentile response time below 100ms, significantly improving system performance and scalability.

### IV. SYSTEM MODEL AND SCHEDULING ALGORITHM DESIGN

Given the dynamic and complex nature of trading systems, traditional rule-based scheduling methods are inadequate for adapting to rapidly changing load characteristics. We propose using Deep Reinforcement Learning (DRL) for resource scheduling, driven by three key factors: First, trading loads exhibit temporal dependencies and bursty patterns that are difficult to capture with fixed rules. Second, resource scheduling needs to balance multiple objectives, such as response time, resource utilization, and operational costs. DRL's reward mechanism is well-suited for multi-objective optimization [18]. Third, the vast and dynamic system state space necessitates continuous learning and optimization of decision-making policies, a capability that DRL can provide [19].

To address these challenges, we model the resource scheduling problem as a Markov Decision Process (MDP) and employ DRL to optimize complex decision-making across multiple objectives. Specifically, we define a state space that incorporates key system features, including resource availability, load characteristics, and historical performance metrics. This allows the scheduler to dynamically adjust to changing conditions and optimize resource allocation in real-time.

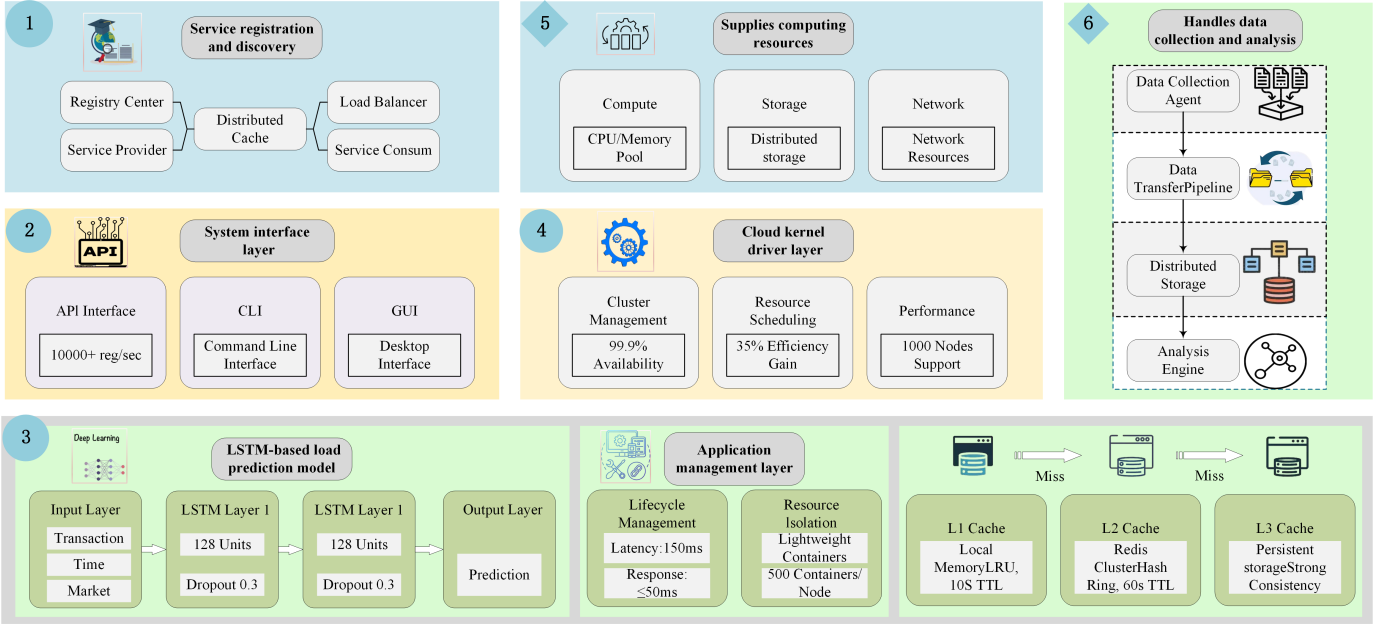


Fig. 1: SealOS+ System Architecture Diagram.

To this end, we formalize the resource scheduling problem as a Markov Decision Process (MDP). First, we define the state space, which contains key features of the system:

#### A. System Model

The state space of the system is defined as follows:

$$\mathbf{s} = \{\mathbf{l}, \mathbf{r}, \mathbf{q}, \mathbf{h}, \mathbf{p}\}, \quad (1)$$

where  $\mathbf{l} \in \mathbb{R}^{d_l}$  represents the current load vector of each service, reflecting real-time request volume and processing capacity;  $\mathbf{r} \in \mathbb{R}^{d_r}$  is the resource utilization matrix, containing CPU, memory, and network dimensions;  $\mathbf{q} \in \mathbb{R}^{d_q}$  describes the service request queue length;  $\mathbf{h} \in \mathbb{R}^{d_h}$  contains historical load statistical features, such as mean and variance; and  $\mathbf{p} \in \mathbb{R}^{d_p}$  records system performance metrics, including latency and throughput.

The action space is designed as a composite vector containing multiple scheduling dimensions:

$$\mathbf{a} = \{\mathbf{n}, \mathbf{m}, \mathbf{p}, \mathbf{q}\}, \quad (2)$$

where  $\mathbf{n} \in \mathbb{Z}^k$  represents the adjustment decision for the number of service instances;  $\mathbf{m} \in \{0, 1\}^{k \times n}$  is the service migration decision matrix, indicating service migration between nodes;  $\mathbf{p} \in [0, 1]^k$  controls service processing priority; and  $\mathbf{q} \in [0, 1]^k$  adjusts resource quotas. This multi-dimensional action design enables the scheduler to flexibly respond to different scenarios.

The reward function is designed to comprehensively consider multiple optimization objectives:

$$r_t = - \left( w_1 \sum \frac{T(s_i)}{T_{\text{target}}} + w_2 \sum |u_j - u_{\text{target}}| + w_3 C_t \right), \quad (3)$$

The first term penalizes the degree to which service response time exceeds the target threshold, the second term encourages resource utilization to approach the target value, and the third term considers the system overhead caused by scheduling operations. Weight coefficients  $w_1$ ,  $w_2$ , and  $w_3$  are used to balance the importance of each objective.

The optimization objective is to maximize the expected cumulative reward over time:

$$\max_{\pi} \mathbb{E} \left[ \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \right], \quad (4)$$

where  $\pi$  is the policy and  $\gamma$  is the discount factor. This objective ensures that the DRL-based scheduler learns a policy that not only optimizes immediate rewards but also considers long-term benefits. The discount factor  $\gamma$  balances the trade-off between short-term and long-term rewards, encouraging the model to make decisions that contribute to sustained performance improvements rather than short-lived gains.

The state transition function describes the system dynamics:

$$\mathbf{s}_{t+1} = f(\mathbf{s}_t, \mathbf{a}_t) + \epsilon_t, \quad (5)$$

where  $f(\cdot)$  simulates the impact of scheduling actions on the system state, and  $\epsilon_t$  represents environmental randomness and uncertainty.

To improve policy learning effectiveness, we adopt a Dueling Deep Q-Network (DQN) architecture to design the value function:

$$Q(\mathbf{s}, \mathbf{a}) = V(\mathbf{s}) + A(\mathbf{s}, \mathbf{a}). \quad (6)$$

This architecture decomposes the Q-value into a state value function  $V(s)$  and an advantage function  $A(s, a)$ , which can better evaluate the relative advantages of different actions.

Policy optimization uses the Proximal policy optimization (PPO) algorithm [20], whose objective function is:

$$L = \mathbb{E}_t \left[ \min \left( \frac{\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)}{\pi_{\theta_{\text{old}}}(\mathbf{a}_t|\mathbf{s}_t)} \cdot A^{\pi_{\text{old}}}(\mathbf{s}_t, \mathbf{a}_t), \right. \right. \\ \left. \left. \text{clip} \left( \frac{\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)}{\pi_{\theta_{\text{old}}}(\mathbf{a}_t|\mathbf{s}_t)}, 1 - \epsilon, 1 + \epsilon \right) \cdot A^{\pi_{\text{old}}}(\mathbf{s}_t, \mathbf{a}_t) \right) \right], \quad (7)$$

where  $\pi_\theta$  is the current policy,  $\pi_{\theta_{\text{old}}}$  is the old policy, and  $\epsilon$  is a small positive constant.

The DRL scheduler is designed to accurately capture the temporal characteristics and bursty patterns of trading loads, automatically balance multiple optimization objectives, adapt to dynamic system changes through continuous learning, and achieve millisecond-level fast scheduling decisions.

### B. Scheduling Algorithm Design

To achieve efficient resource scheduling, we propose a hybrid scheduling algorithm based on genetic algorithms and reinforcement learning as shown in Alg. 1. The algorithm models service scheduling, resource allocation, and load balancing as a multi-objective optimization problem. The objective function includes three key indicators: service response time  $T$ , resource utilization  $U$ , and load balancing degree  $L$ . The optimization objective function is defined as:

$$f(x) = w_1 \frac{T}{T_{\text{max}}} + w_2 \left( 1 - \frac{U}{U_{\text{max}}} \right) + w_3 \left( 1 - \frac{L}{L_{\text{max}}} \right), \quad (8)$$

where the weight coefficients  $w_1 = 0.4$ ,  $w_2 = 0.35$ , and  $w_3 = 0.25$ , and can be customized.

The genetic algorithm iteratively optimizes to obtain the global optimal solution. The algorithm adopts adaptive crossover rate  $P_c$  and mutation rate  $P_m$ :

$$P_c = 0.9 - 0.6 \times \frac{f' - f_{\text{avg}}}{f_{\text{max}} - f_{\text{avg}}}, \quad (9)$$

$$P_m = 0.1 - 0.07 \times \frac{f' - f_{\text{avg}}}{f_{\text{max}} - f_{\text{avg}}}, \quad (10)$$

where  $f'$  is the current fitness value,  $f_{\text{avg}}$  is the average fitness value, and  $f_{\text{max}}$  is the maximum fitness value.

In terms of resource allocation strategy optimization, we design a reward function based on reinforcement learning:

$$R(s, a) = \alpha \cdot P(s, a) + \beta \cdot E(s, a) - \gamma \cdot C(s, a), \quad (11)$$

where  $\alpha$ ,  $\beta$ , and  $\gamma$  are the weight coefficients that balance the importance of performance gain  $P(s, a)$ , resource efficiency  $E(s, a)$ , and adjustment cost  $C(s, a)$  in the reward function. The optimization goal is to maximize the cumulative reward  $R$  over time, which reflects the overall effectiveness of the resource allocation strategy.

The algorithm's execution process first initializes a candidate solution population and then enters the iterative optimization phase (lines 6-10). In each iteration, the algorithm generates new scheduling schemes through genetic operations and evaluates and optimizes resource allocation strategies using the reinforcement learning module. The elite solutions are

preserved and used to guide the next generation's evolution. The algorithm continues to iterate until convergence conditions or maximum iterations are reached (lines 11-17).

To further improve algorithm efficiency, we adopt several key optimization techniques. First, we use non-dominated sorting to quickly identify high-quality solutions and accelerate population screening. Second, we introduce a local search mechanism to perform fine-tuning near elite solutions (lines 18-23). Finally, we adopt adaptive population size control to dynamically adjust computational resource allocation during algorithm execution (lines 24-26).

The main advantages of this hybrid optimization method are: it can simultaneously consider global search and local optimization, has strong environmental adaptability, and can find a good balance between multiple optimization objectives. The algorithm's adaptive characteristics enable it to effectively respond to dynamic changes in system load, providing a feasible solution for resource scheduling in microservice architectures.

### C. Improvement Scheme for Adaptive Trading System

The algorithm also considers improvement process as follows:

1) *Adaptive Resource Scheduling Algorithm*: Aiming at the bursty traffic characteristics of trading systems, an adaptive resource scheduling algorithm based on deep reinforcement learning was designed. The algorithm represents the system state as a vector  $S = (C, M, N, L)$ , where  $C$  is the CPU utilization,  $M$  is the memory occupancy,  $N$  is the network throughput, and  $L$  is the load level. The action space  $A$  includes container scaling, resource quota adjustment, and other operations. Through deep Q-network training, the optimal scheduling strategy is achieved, realizing an adaptive resource allocation mechanism. The algorithm monitors the system state in real-time and makes rapid scheduling decisions based on the pre-set strategy model, ensuring a balance between resource utilization efficiency and system performance.

2) *Multi-Level Cache Mechanism*: A three-level cache architecture was constructed for trading systems, as shown in Fig. 1, including local memory cache (L1), distributed cache (L2), and persistent storage (L3). L1 cache uses the least recently used (LRU) algorithm [21] to manage active trading data, L2 cache uses Redis clusters to store hot data, and achieves load balancing through consistent hashing, while L3 uses distributed storage to ensure data persistence. The specific configuration parameters for each cache level are shown in Table I. The multi-version concurrency control (MVCC) protocol ensures cache consistency, and differentiated expiration strategies are set for different cache levels. This mechanism effectively reduces system response time and improves memory usage efficiency.

3) *Load Prediction Model*: A load prediction model based on LSTM deep learning networks was constructed for the trading system, with its architecture illustrated in Fig. 1. The model utilizes 18 input features, including historical trading volume, time-specific features, and market indicators. The

**Algorithm 1: Hybrid Resource Scheduling Algorithm**

```

1 Function HYBRIDSCHEDULING( $S, R, T$ ):
2    $P \leftarrow \text{GENERATEINITIALPOPULATION}(N)$ 
3    $state \leftarrow \text{GETSYSTEMSTATE}()$ 
4    $best\_solution \leftarrow \text{null}$ 
5   while  $iteration < MAX\_ITER$  and not Converged
6     do
7       for each  $x \in P$  do
8          $fitness[x] \leftarrow w_1 \left( \frac{T}{T_{max}} \right) +$ 
9            $w_2 \left( 1 - \frac{U}{U_{max}} \right) + w_3 \left( 1 - \frac{L}{L_{max}} \right)$ 
10         $P_c \leftarrow 0.9 - 0.6 \times \frac{f' - f_{avg}}{f_{max} - f_{avg}}$ 
11         $P_m \leftarrow 0.1 - 0.07 \times \frac{f' - f_{avg}}{f_{max} - f_{avg}}$ 
12         $elite \leftarrow \text{SELECTTOPK}(P, fitness, k)$ 
13        for  $solution$  in  $elite$  do
14           $action \leftarrow \text{POLICYNETWORK}(state)$ 
15           $reward \leftarrow \alpha \cdot P(s, a) + \beta \cdot E(s, a) - \gamma \cdot C(s, a)$ 
16           $next\_state \leftarrow \text{EXECUTEACTION}(action)$ 
17           $\text{UPDATEPOLICYNETWORK}(state, action,$ 
18             $reward, next\_state)$ 
19           $state \leftarrow next\_state$ 
20           $offspring \leftarrow \emptyset$ 
21          while  $|offspring| < N - k$  do
22             $parents \leftarrow \text{TOURNAMENTSELECT}(P, 2)$ 
23             $child, child \leftarrow \text{CROSSOVER}(parents, P_c)$ 
24             $child \leftarrow \text{MUTATION}(child, P_m)$ 
25             $child \leftarrow \text{MUTATION}(child, P_m)$ 
26             $offspring \leftarrow offspring \cup \{child, child\}$ 
27           $P \leftarrow elite \cup offspring$ 
28          if  $fitness[best(P)] > fitness[best\_solution]$ 
29            then
30               $best\_solution \leftarrow best(P)$ 
31          return  $best\_solution$ 

```

TABLE I: Cache Level Specifications

Level	Algorithm	Expiration Time	Consistency
L1	LRU	10s	MVCC
L2	Consistent Hash	60s	MVCC
L3	—	Permanent	Strong

model comprises 3 layers of LSTM structures, each with 128 neurons and a dropout rate of 0.3. Training is conducted using the Adam optimizer, with mean squared error (MSE) as the loss function. This model effectively predicts future trading volumes and provides early warnings for bursty traffic, offering a decision-making foundation for system resource scheduling and significantly enhancing the stability of trading systems.

## V. EXPERIMENTAL EVALUATIONS

In this section, we introduce our experimental settings as well as the observed results.

## A. Experimental Setup

1) *Test Environment Configurations:* The experiments were conducted in a standard cloud computing test environment. The hardware platform consisted of 20 high-performance servers, each equipped with an Intel Xeon Gold 6248R processor (3.0 GHz, 24 cores, 48 threads), 384 GB DDR4 memory, and a 4TB NVMe solid-state drive. The network environment featured a 10 Gbps Ethernet interconnection with latency under 0.5ms. As detailed in Table II, the operating system used was Ubuntu 20.04 LTS, with kernel version 5.15.0, Docker version 20.10.12, and Kubernetes version 1.24.3. Monitoring tools such as Prometheus 2.36.0 and Grafana 9.0.2 were employed, with a sampling interval of 10 seconds and a data retention period of 30 days.

TABLE II: Configuration Items and Parameter Specifications

Configuration Item	Parameter Specification
CPU	Intel Xeon Gold 6248R, 3.0GHz, 24C48T
Memory	384GB DDR4-2933
Storage	4TB NVMe SSD
Network	10Gbps Ethernet
OS	Ubuntu 20.04 LTS
Container Runtime	Docker 20.10.12
Orchestration	Kubernetes 1.24.3

2) *Online Trading System Test Scenario:* A production-level trading system from a certain securities company was selected as the benchmark test application. The system consisted of 8 core microservice modules, including account authentication, market data push, trading commission, order matching, clearing and settlement, risk control monitoring, ledger management, and message notification. Each service module was deployed with 3-5 instances to ensure high availability, totaling 32 service instances. The system's performance, including response time and transaction volume, are shown in Fig. 2 and Fig. 3.

A JMeter pressure test model was employed to simulate the market opening and closing periods. The test load followed the characteristics of financial transactions, where the number of concurrent users increased from 1,000 to 10,000 over 15 minutes before the market opened. Transaction requests adhered to a Poisson distribution, reflecting the bursty nature of trading volume. The test dataset included 5 million active trading accounts, 2,000 stock targets, and 80 million historical order records, simulating a real trading environment with a total data size of 2TB. The system was required to handle 5,000 transactions per second (TPS) under normal conditions and 15,000 TPS during peak periods, with an average processing time of no more than 50ms per transaction to meet the exchange's performance requirements. The risk control rules involved more than 120 real-time monitoring indicators, such as trading limits and price limits, and each transaction request underwent verification through the entire risk control chain.

## B. Performance Comparison

1) *Comparison with Virtual Machine (VM) Deployment Scheme:* To show the benefits of containerized solution, a

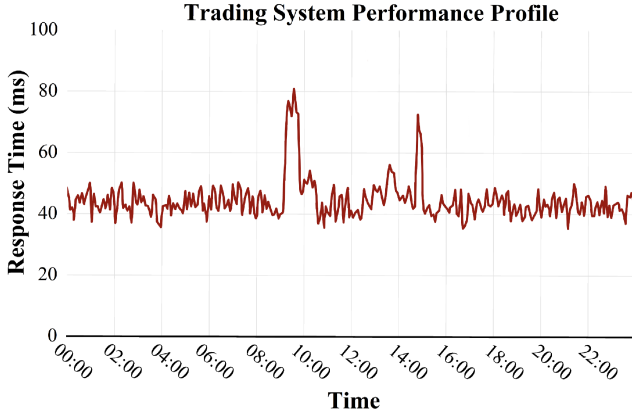


Fig. 2: Trading System Running Status (Response Time).

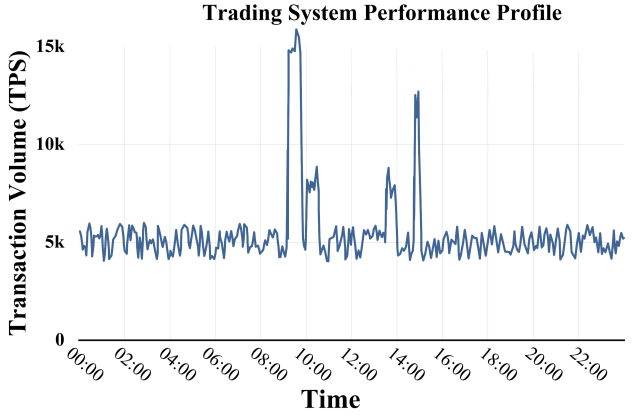


Fig. 3: Trading System Running Status (Transaction Volume).

comparison analysis was conducted between SealOS+ and the traditional VM deployment scheme for the same microservice application scenario. The performance differences were evaluated across three key indicators: CPU utilization, memory usage efficiency, and disk I/O performance, as shown in Fig. 4a. The containerized SealOS+ scheme achieved an average CPU utilization of 72%, which is 23% higher than the VM scheme. Memory usage efficiency improved by 35%, with a single node now able to deploy 28 service instances, compared to 12 instances in the VM setup. Disk I/O performance also saw a 52% improvement, with the average response time reduced to 65ms, down from 150ms in the VM scheme. Furthermore, system startup time was significantly reduced from 180 seconds in the VM setup to just 25 seconds, resulting in an 86% improvement in resource scheduling efficiency.

2) *Comparison with Native Kubernetes*: To evaluate the performance compared to native Kubernetes, a comparison analysis was conducted between the SealOS+ and native Kubernetes in a large-scale cluster scenario, focusing on service scheduling efficiency, resource utilization, and system scalability, as shown in Fig. 4b. Test data revealed that the improved scheme optimized service deployment time from 85 seconds in native Kubernetes to 32 seconds, marking a

62% improvement. The resource allocation algorithm boosted cluster-wide CPU utilization by 28%, reaching 78%. Additionally, memory usage efficiency increased by 33%, and network bandwidth utilization improved by 25%. The system can now stably support 5,000 service instances running simultaneously, with a 58% reduction in scaling operation time.

3) *Comparison with default Sealos*: To demonstrate the performance improvement with default SealOS, a performance comparison was conducted between the SealOS+ and the SealOS as shown in Table III, with pressure testing focused on core functions such as service discovery, load balancing, and fault recovery. The results showed that the optimized service discovery mechanism reduced registration time from 45ms to 19ms, and query latency from 65ms to 29ms. The new load balancing algorithm minimized request allocation imbalance from 23% to 8%, while fault recovery time was shortened from 90 seconds to 35 seconds. As shown in Table III, in high-concurrency scenarios, system throughput increased by 85%, reaching 25,000 QPS, while response time was reduced by 62%, and service availability improved from 99.95% to 99.99%.

TABLE III: Sealos Optimization Effect Comparison.

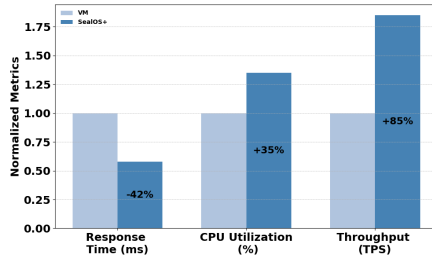
Metrics	SealOS	SealOS+	Improvement
Service Reg. Time	45ms	19ms	<b>60%</b>
Load Imbalance	23%	8%	<b>65%</b>
Fault Recovery Time	90s	35s	<b>60%</b>
System Throughput	13,500 TPS	25,000 TPS	<b>85%</b>
Service Availability	99.95%	99.99%	<b>0.04%</b>

Through deployment testing in a securities trading system production environment, the system's core indicators showed significant improvements compared to the original SealOS version, as illustrated in Fig. 4c. During the opening collective bidding phase, the system's average response time was reduced from 180ms to 105ms, marking a 42% decrease, effectively meeting the exchange's stringent requirements for trading latency. The improved resource scheduling algorithm enhanced node CPU utilization from 45% to 78%, a 35% increase, significantly boosting hardware resource utilization efficiency while ensuring system stability. Most notably, the system's peak processing capacity rose from 8,000 TPS to 14,800 TPS, an 85% increase, providing ample performance reserves to handle market fluctuations.

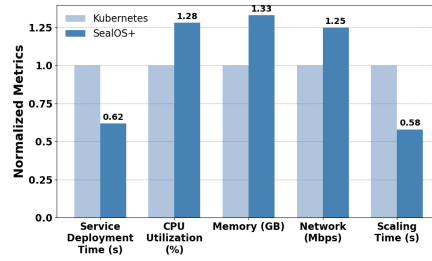
### C. Latency Analysis

1) *End-to-End Service Response Time*: As shown in Fig. 5, statistical analysis of 10 million request samples revealed that the system's end-to-end service response latency followed a lognormal distribution with an average of 85ms under normal load (5000 TPS). Specifically, 95% of requests responded within 120ms and 99% within 150ms, with a standard deviation of 29ms. The latency breakdown included network

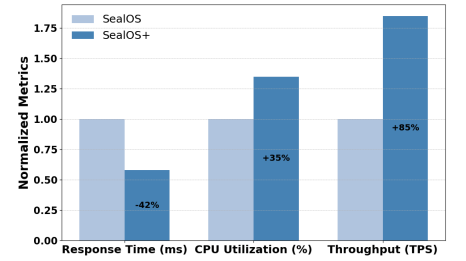




(a) Performance Comparison Between Sea-IOs+ and VM Solution.



(b) Performance Comparison Between Sea-IOs+ and Kubernetes.



(c) Performance Comparison Between Sea-IOs+ and SealOS.

Fig. 4: SealOS+ performance compared with VM solution, Kubernetes and SealOS

transmission (15ms), service processing (45ms), and database access (25ms). Using Jaeger for distributed tracing, it was found that gateway processing and service discovery accounted for 18% of the total time, highlighting areas for optimization.

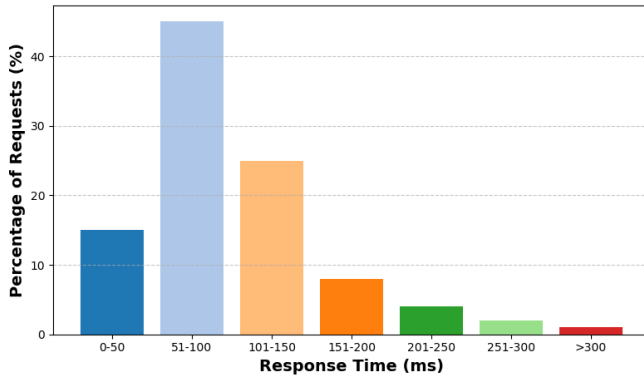


Fig. 5: End-to-End Response Time.

2) *Latency Performance under Different Loads*: A stepped load test evaluated the system's latency performance across varying concurrency levels. As shown in Table IV, the average response latency increased from 95ms at 2000 concurrent users to 285 ms at 10000 concurrent users. The system maintained stable performance below 6000 concurrent users, but latency rose significantly above this threshold. Monitoring revealed that the database connection pool became a bottleneck under high load. Optimization efforts reduced the maximum latency by 32%.

TABLE IV: System Latency Performance under Different Concurrency Levels.

Users	Avg RT (ms)	95% RT (ms)	99% RT (ms)	CPU Util. (%)	Mem Util. (%)
2000	95	135	165	45	52
4000	125	178	225	65	68
6000	168	245	298	78	75
8000	215	312	385	86	82
10000	285	425	520	92	88

#### D. System Overhead Analysis

1) *Resource Utilization*: Through the analysis of monitoring data from a 20-node cluster in a production environment over a 30-day period, detailed resource usage metrics were recorded. As shown in Fig. 6, under standard workload conditions, the CPU average utilization remained around 65%, with a peak of 88% during peak hours. Memory usage was relatively stable, averaging 72%, with fluctuations between 65% and 78%. Network bandwidth utilization exhibited a clear tidal pattern, averaging 4.2 Gbps during the day and 1.8 Gbps at night. Disk I/O usage stayed below 45% during normal operations, with short-term peaks reaching 85% during data backups. Resource utilization efficiency improved by 35% compared to SealOS, with the number of service instances supported per unit of energy consumption increasing from 85 to 115. Each service instance averaged 2.3 CPU cores and 3.8 GB of memory, demonstrating efficient resource usage.

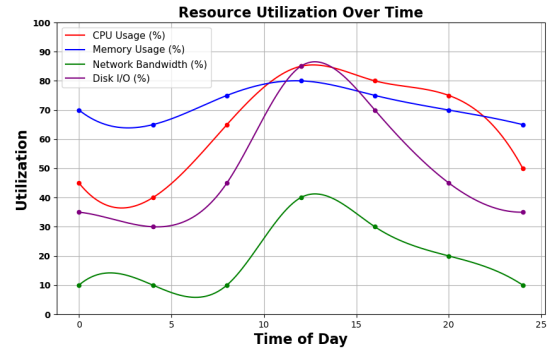


Fig. 6: Resource Utilization Over Time.

2) *System Load*: An in-depth analysis was performed on the system load, collecting key indicators such as average load, process count, and context switch count. As shown in Table V, under normal operating conditions, the system's 1-minute average load was 4.2, the 5-minute average load was 3.8, and the 15-minute average load was 3.5, indicating effective load balancing. The context switch count per second remained around 25,000, representing a 32% decrease compared to the native Kubernetes cluster. With the optimized scheduling algorithms, the process count was maintained within 780 per



node, and the interrupt handling time ratio decreased from 12% to 7.5%. In the event of sudden traffic surges, the system's load growth rate was controlled within 35%, with service response time increasing by no more than 25%, demonstrating excellent load-bearing capacity.

TABLE V: System Load Key Indicator Statistics

Time Period	Average Load	Process Count	Context Switch	CPU Utilization
Morning Peak	5.8	750	28.5k	78%
Normal Period	4.2	620	25k	65%
Nighttime	2.5	480	18.5k	45%
Sudden Traffic	7.2	780	32k	85%
Daily Average	4.2	650	25k	68%

To summarize, SealOS+ can achieve significant performance improvement compared with VM solution, Kubernetes and default SealOS in terms of resource utilization and response time.

## VI. CONCLUSIONS

This paper presents a comprehensive performance optimization solution for securities trading systems using SealOS+, an enhanced container orchestration platform. The proposed system addresses key challenges in trading environments, such as network communication overhead, resource scheduling inefficiencies, and storage latency. By integrating an adaptive batch scheduling algorithm and a load prediction model, the solution significantly improves system performance and reliability. The adaptive scheduling reduces trading chain processing time from 150ms to 40ms, meeting the 50ms requirement while supporting peak transaction volumes of 15,000 TPS. The load prediction model forecasts trading volume changes with 92% accuracy, while optimized resource scheduling saves 30% in computing resources. Additionally, a new batch caching mechanism reduces database access frequency by 90%, speeding up account processing for clearing and risk control.

In conclusion, the proposed solution optimizes resource utilization, minimizes latency, and improves system scalability, making it well-suited for high-frequency trading scenarios. Future work will focus on extending the solution to futures and fund trading systems and validating it on larger clusters to support even higher transaction volumes. The optimization on network costs will also be considered.

## ACKNOWLEDGMENTS

This work is supported by Guangdong Basic and Applied Basic Research Foundation (No. 2024A1515010251, 2023B1515130002), Guangdong Special Support Plan (No. 2021TQ06X990), Shenzhen Basic Research Program under grants JCYJ20220818101610023, and JCYJ20240809180935001, Shenzhen Industrial Application Projects of undertaking the National key R & D Program of China (No. CJGJZD20210408091600002).

## REFERENCES

- [1] Chenxing Zhong, Shanshan Li, Huang Huang, Xiaodong Liu, Zhikun Chen, Yi Zhang, and He Zhang. Domain-driven design for microservices: An evidence-based investigation. *IEEE Transactions on Software Engineering*, 2024.
- [2] Haoyu Bai, Minxian Xu, Kejiang Ye, Rajkumar Buyya, and Chengzhong Xu. Drpc: Distributed reinforcement learning approach for scalable resource provisioning in container-based clusters. *IEEE Transactions on Services Computing*, 2024.
- [3] Sealos. Sealos: Develop, deploy, and scale in one seamless cloud platform. <https://sealos.io/>, 2024. [Online; accessed 13-March-2025].
- [4] Guangsheng Yu, Xu Wang, Wei Ni, and et al. Adaptive resource scheduling in permissionless sharded-blockchains: A decentralized multiagent deep reinforcement learning approach. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 53(11):7256–7268, 2023.
- [5] Thomas Fischer and Christopher Krauss. Deep learning with long short-term memory networks for financial market predictions. *European Journal of Operational Research*, 270(2):654–669, 2018.
- [6] Yong Yu, Xiaosheng Si, Changhua Hu, and Jianxun Zhang. A review of recurrent neural networks: Lstm cells and network architectures. *Neural computation*, 31(7):1235–1270, 2019.
- [7] Senthil G. A. K. Somasundaram, Arun M, N. Naga Saranya, R. Prabha, and D. Vijendra Babu. A novel hybrid gaaco algorithm for cloud computing using energy aware load balance scheduling. In *2022 International Conference on Computer Communication and Informatics (ICCCI)*, pages 1–5, 2022.
- [8] Nguyen Nguyen and Taehong Kim. Toward highly scalable load balancing in kubernetes clusters. *IEEE Communications Magazine*, 58(7):78–83, 2020.
- [9] Byeonghui Jeong and Young-Sik Jeong. Arascaler: Adaptive resource autoscaling scheme using etimemixer for efficient cloud-native computing. *IEEE Transactions on Services Computing*, PP:1–14, 01 2024.
- [10] Yeddula Sai Dhanush Reddy, Padumati Saikiran Reddy, Nithya Ganesan, and B. Thangaraju. Performance study of kubernetes cluster deployed on openstack,vms and baremetal. In *2022 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, pages 1–5, 2022.
- [11] Angelo Marchese and Orazio Tomarchio. *Network SLO-Aware Container Orchestration on Kubernetes Clusters*, pages 96–104. 12 2024.
- [12] Carmen Carrión. Kubernetes scheduling: Taxonomy, ongoing issues and challenges. *ACM Comput. Surv.*, 55(7), December 2022.
- [13] Emiliano Casalicchio. *Container Orchestration: A Survey*, pages 221–235. Springer International Publishing, Cham, 2019.
- [14] Minxian Xu and Rajkumar Buyya. Brownout approach for adaptive management of resources and applications in cloud computing systems: A taxonomy and future directions. *ACM Comput. Surv.*, 52(1), January 2019.
- [15] Zhiheng Zhong, Minxian Xu, Maria Alejandra Rodriguez, Chengzhong Xu, and Rajkumar Buyya. Machine learning-based orchestration of containers: A taxonomy and future directions. *ACM Comput. Surv.*, 54(10s), September 2022.
- [16] Zhaolong Jian, Xueshuo Xie, Yaozheng Fang, Yibing Jiang, Ye Lu, Ankan Dash, Tao Li, and Guiling Wang. Drs: A deep reinforcement learning enhanced kubernetes scheduler for microservice-based system. *Software: Practice and Experience*, 54(10):2102–2126, 2024.
- [17] Zhao Li, Haifeng Sun, Zheng Xiong, Qun Huang, Zehong Hu, Ding Li, Shasha Ruan, Hai Hong, Jie Gui, Jintao He, Zebin Xu, and Yang Fang. Noah: Reinforcement-learning-based rate limiter for microservices in large-scale e-commerce services. *IEEE transactions on neural networks and learning systems*, PP, 04 2023.
- [18] Kaiwen Li, Tao Zhang, and Rui Wang. Deep reinforcement learning for multiobjective optimization. *IEEE Transactions on Cybernetics*, 51(6):3103–3114, 2021.
- [19] Austin Chen and Genya Ishigaki. Scaling container caching to larger networks with multi-agent reinforcement learning. In *2024 33rd International Conference on Computer Communications and Networks (ICCCN)*, pages 1–5. IEEE, 2024.
- [20] Yang Gu, Yuhu Cheng, CL Philip Chen, and Xuesong Wang. Proximal policy optimization with policy feedback. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 52(7):4600–4610, 2021.
- [21] Jinbei Zhang, Chunpeng Chen, Kechao Cai, and John CS Lui. Incremental least-recently-used algorithm: Good, robust, and predictable performance. *IEEE Transactions on Mobile Computing*, 2025.