Journal of Computer Technology and Software

ISSN:2998-2383

Vol. 4, No. 4, 2025

High-Concurrency Distributed Task Scheduling with Fault Tolerance and Load Balancing

Anneliese Wren[1], Callum Hargreaves[2], Elodie Farnsworth[3]

[1]University of Southern Maine, Portland, Maine, USA

[2]University of Southern Maine, Portland, Maine, USA

[3]University of Southern Maine, Portland, Maine, USA

*Corresponding Author: Anneliese Wren; awren932@usm.edu

Abstract: In modern distributed computing environments, scheduled task scheduling systems play a key role in data synchronization, batch processing, and automated operation and maintenance. However, traditional single-machine task scheduling methods face problems such as single point failure, task concurrency bottlenecks, and scheduling instability, making it difficult to meet high concurrency and high availability business requirements. To address these challenges, this study designed and implemented a highly available distributed scheduled task scheduling system. The system uses multi-machine hot standby and lock contention triggering mechanisms to ensure reliable triggering of tasks in a distributed environment, and combines ZooKeeper for task coordination to avoid duplicate execution problems. In addition, the system optimizes computing resource utilization through dynamic load balancing strategies, and uses asynchronous RPC interactions to improve task scheduling throughput. In order to verify the stability of the system, this study conducted integration tests and high availability tests. Experimental results show that the system can still ensure the normal operation of task scheduling in the case of multi-node failures, improving the reliability of task execution. This study provides an efficient and stable distributed task scheduling solution for enterprise-level applications, which can be widely used in the Internet, finance, telecommunications and other industries, and has important engineering value and theoretical significance.

Keywords: Distributed computing, high availability, task scheduling, load balancing

1. Introduction

In today's highly information-driven society, scheduled task scheduling systems have become an indispensable component of enterprise software architectures, widely applied in scenarios such as data synchronization, batch processing, and automated operations and maintenance. However, with the rapid advancement of cloud computing, big data, and microservices architecture, the traditional single-machine scheduling model faces increasing challenges, including single points of failure, task concurrency bottlenecks, and unstable scheduling. Particularly in large-scale distributed environments, enterprises must ensure that tasks operate stably under high concurrency and heavy workloads while maintaining robust fault tolerance. Therefore, researching the design and implementation of a high-availability distributed scheduled task scheduling system holds significant theoretical and practical value[1].

Traditional scheduled task scheduling methods primarily rely on operating system-level tools such as Crontab or application-layer frameworks like Quartz. While these tools perform well for small-scale task management, their limitations become increasingly apparent when addressing the complex demands of enterprise applications. For instance, Crontab is dependent on a single server, meaning that if the server fails, task scheduling will be entirely disrupted. Similarly, Quartz offers a rich set of task management features but still encounters challenges in distributed environments, such as database storage bottlenecks and complex task coordination[2]. To overcome these limitations, a variety of distributed task scheduling frameworks have emerged in recent years, including Apache Airflow, XXL-JOB, and ElasticJob.

Although these frameworks address some of the shortcomings of single-machine scheduling, they still suffer from issues such as task drift, inefficient load balancing, and complex dependency management. Thus, investigating a more efficient and reliable distributed scheduled task scheduling system remains a promising research direction[3].

The core objective of a high-availability distributed scheduled task scheduling system is to ensure scheduling stability, scalability, and fault tolerance. In a distributed environment, several key technical challenges must be addressed, including a highly reliable task-triggering mechanism (to prevent task loss or redundant execution), dynamic task load balancing (to allocate tasks efficiently based on computing node workload), and task state management and recovery (to enable automatic migration or re-execution in case of node failures). Additionally, to enhance execution efficiency, the system should support task partitioning and parallel execution, allowing large-scale computational tasks to fully

leverage cluster resources and improve overall throughput. A well-designed distributed scheduled task scheduling system not only enhances the operational efficiency of enterprise applications but also reduces maintenance costs and strengthens business continuity.

In recent years, with the evolution of distributed computing and cloud-native architectures, an increasing number of scheduling systems have adopted decentralized architectures and integrated distributed consensus protocols (e.g., Raft, Paxos) to ensure scheduling stability. For example, Kubernetes CronJob enables container-based task scheduling with dynamic resource scaling capabilities, but its coarse-grained scheduling approach makes it difficult to meet high-precision scheduling requirements. Meanwhile, ElasticJob employs a sharding-based scheduling mechanism to enhance distributed execution capabilities, but its scheduling strategies still require optimization under extreme workloads. Therefore, by integrating the advantages of existing scheduling frameworks and optimizing for high-concurrency and high-availability scenarios, developing a more comprehensive distributed scheduled task scheduling system can not only contribute new insights to academic research but also provide efficient solutions for enterprise applications[4].

This study aims to design and implement a high-availability distributed scheduled task scheduling system, focusing on core functionalities such as reliable task triggering, dynamic load balancing, task state management, and fault recovery. By incorporating key technologies such as distributed storage, distributed consensus protocols, and task-sharding mechanisms, this research seeks to construct a task scheduling architecture that is highly available, scalable, and efficient, providing enterprises with a stable and reliable task management solution. The outcomes of this research can be widely applied to various Internet business scenarios and extended to industries such as finance, telecommunications, and intelligent manufacturing, offering robust support for both theoretical research and engineering practices in distributed task-scheduling.

2. Related Work on Distributed and Intelligent Task Scheduling

Recent advances in distributed computing and intelligent scheduling have significantly influenced the development of high-availability task scheduling systems. Research has increasingly turned toward integrating machine learning, particularly reinforcement learning, into distributed scheduling frameworks to enhance efficiency and scalability. Wang [5] proposed a federated learning-based resource optimization framework that emphasizes communication efficiency and adaptive task scheduling, providing important theoretical support for decentralized task coordination. Similarly, Deng [6] explored traffic scheduling in data centers using reinforcement learning, demonstrating the potential of adaptive learning strategies in managing complex network environments.

Reinforcement learning has also been applied in specific domains such as operating systems and IoT. Sun et al. [7] introduced a Double DQN-based method for dynamic OS scheduling, showcasing how task optimization can benefit from learning-based decision mechanisms. He et al. [8] combined Deep Q-Networks with edge-based coordination for IoT scheduling, further validating the feasibility of intelligent scheduling strategies in heterogeneous environments.

Trust-aware mechanisms and policy learning have been studied by Ren et al. [9], who designed a distributed network traffic scheduler using trust-constrained reinforcement learning, effectively addressing reliability and trust in multi-agent scheduling scenarios. Complementing this, Li et al. [10] applied contrastive learning in unsupervised fraud detection, highlighting methodological innovations that could enhance anomaly detection and task verification in distributed scheduling systems.

Transformers and attention mechanisms have also become increasingly relevant in system optimization and anomaly detection. Xu [11] employed transformer models for structural anomaly detection in video integrity tasks, while Liang [12] proposed a graph attention-based recommendation framework for sparse interactions — both illustrating how attention mechanisms can be adapted to task prioritization and dependency resolution. Additionally, Guo et al. [13] explored self-supervised Vision Transformers, suggesting the utility of unsupervised representation learning in task classification and context understanding.

Cross-domain and multimodal deep learning techniques offer insights for managing heterogeneous task environments. Zhu [14] developed a spatial-channel attention model for cross-domain recommendation, which could be adapted for distributed task routing and context-aware scheduling. In parallel, Li et al. [15] presented a CNN-Transformer model for multimodal classification, showcasing the feasibility of integrating diverse data streams for scheduling decisions in intelligent systems.

Graph neural networks and sequence models further extend this paradigm. Zhang [16] demonstrated the use of graph neural networks for user profiling and anomaly detection in social systems — insights that could inform distributed task dependency modeling. Finally, Sun and Duan [17] utilized BiLSTM for predicting user intent in HCI scenarios, offering methodologies that could support predictive task triggering and adaptive scheduling.

3. Distributed system detailed design

3.1 Detailed design of task scheduling core service subsystem

The Task Scheduling Core Service Subsystem is primarily responsible for task scheduling and triggering management, task state management, client connection state management, and RPC service management. As the central subsystem of the entire system, it plays a crucial role in coordinating the interactions among various system components. Below is the sequence diagram illustrating the interactions between the Task Scheduling Core Service Subsystem and other subsystems. The sequence diagram is shown in Figure 1.

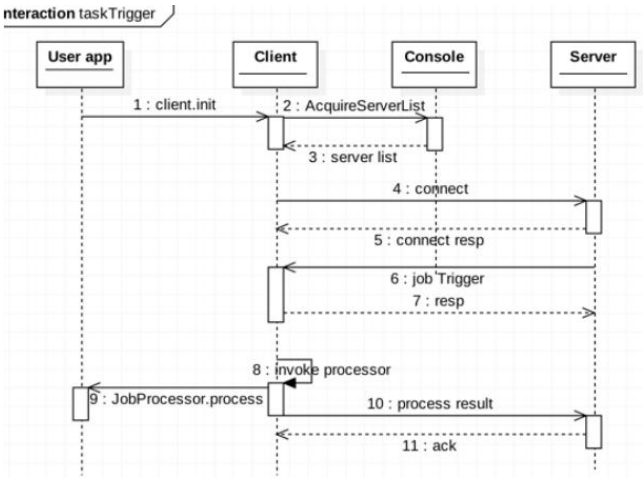

Figure 1. Task trigger scheduling sequence diagram

3.2 Detailed design of task scheduling and control service subsystem

In the design of the task scheduling and control service subsystem, the task grouping management and task creation mechanism are the key links to ensure the efficiency, scalability and high availability of task scheduling. In order to classify and manage permissions for large-scale tasks, the system adopts a task grouping mechanism, and each task must belong to a unique task group. The task group ID adopts a four-segment structure (such as 101-1-2-6601), which represents the cluster ID, ServerGroup ID, the number of task backups and the task number, respectively, to uniquely identify the task and ensure the traceability of the task. In addition, task grouping is also closely related to user permission management and backend server resource allocation. After the user has management permissions for a task group, he can create, modify and schedule tasks under the group. At the same time, the system will automatically assign the appropriate ServerGroup to the task to ensure the stability and scalability of task scheduling.

In terms of task creation management, the system adopts a dynamic task allocation and load balancing mechanism. When creating a task, the system will intelligently allocate tasks according to the current running status of the cluster server. This strategy takes into account the disaster recovery capability across computer rooms, uses the IP information of the server to determine the computer room it belongs to, and preferentially assigns tasks to servers in different computer rooms, thereby enhancing the fault tolerance of the system at the computer room level and avoiding the failure of task triggering due to single-point computer room failure. At the same time, the distribution of tasks adopts a random allocation strategy to ensure that all servers only load part of the tasks, so that the overall task scheduling load of the system is balanced, and supports horizontal expansion to meet the execution requirements of large-scale tasks.

In terms of system architecture, the core components of task management include JobManager, JobAccess, JobAccess4Mysql, ClientGroupManager, ClientGroupAccess and ClientGroupAccess4Mysql. Among them, JobManager and ClientGroupManager are responsible for the management of tasks and groups, and encapsulate the metadata access logic of task scheduling; JobAccess and ClientGroupAccess are data layer interfaces that shield the implementation details of the underlying database and improve the scalability of the system; JobAccess4Mysql and ClientGroupAccess4Mysql provide specific MySQL version implementations to ensure persistent storage and fast retrieval of task data. These components work together to achieve efficient control and data management of task scheduling, and improve the overall stability and maintainability of the system. The group management sequence diagram and task management sequence diagram are shown in Figure 2 and Figure 3.

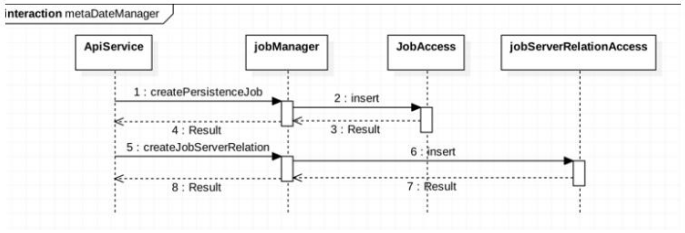

Figure 2. Group management sequence diagram

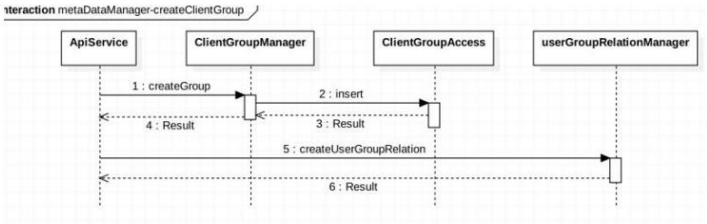

Figure 3. Task management sequence diagram

In the design of the task trigger execution subsystem, the system mainly includes task execution pool management, task execution unit management, business processing Bean management and RPC service management to ensure efficient execution of tasks and decoupling of business logic. The core goal of this subsystem is to provide an efficient and scalable task execution environment, while ensuring the separation of task scheduling and business logic, making the execution of tasks more flexible and stable.

The task scheduling trigger module adopts the JobPool mechanism to establish corresponding JobPools for different types of tasks (Jobs) and uniformly manage the running instances of tasks. The system maintains a mapping table (Map) in memory to store the task instances currently running on the client side in order to efficiently manage the execution status of tasks. Each task instance corresponds to a task execution unit, which integrates a task execution thread pool to process multiple tasks in parallel and improve the throughput and response speed of task execution. The size of the thread pool can be customized by the user according to business needs to adapt to task loads of different sizes.

Inside the task execution unit, the system will call the task processing interface implemented by the user to trigger the process call to the business processing Bean to implement the specific business logic of the task. Through this interface design, the system realizes the decoupling of task scheduling

logic and business processing logic, ensuring that the scheduling module does not directly depend on specific business logic, thereby enhancing the versatility and scalability of the task scheduling system. At the same time, the execution process of the task communicates across nodes through the RPC remote call mechanism, ensuring that the task can be flexibly scheduled to different computing nodes, thereby improving the overall task processing capability and availability of the system.

This chapter mainly carries out a detailed design of the system. Starting from the system requirements, it establishes that the system consists of three subsystems: task scheduling core service subsystem, task scheduling service management and control service subsystem, and task triggering execution subsystem. At the same time, it fully considers the system's reliability and performance requirements, and carries out detailed design of the functional modules of each subsystem.

4. System Testing

4.1 Purpose of the test

In order to verify whether the functional integrity and high availability of the distributed timed task scheduling system meet the design requirements, the system test includes multiple links such as unit testing, integration testing and high availability testing. In the unit testing phase, each functional module of the three subsystems is independently tested through test piles and driver modules to ensure the correctness of the basic functions. After the unit test passes, the system enters the integration testing phase, and each subsystem is debugged in the test environment to verify the stability and correctness of their collaborative work. Finally, in order to evaluate the high availability of the system, in the high availability testing phase, by simulating different types of system failures, the recovery capability and stability of task scheduling under abnormal conditions are observed to ensure that the system can meet the expected high availability standards.

4.2 Test environment

In terms of test environment configuration, both the server and the client are deployed on Google Cloud virtual machines to ensure the stability and reproducibility of the test environment. The server runs the CentOS 7.2 64-bit operating system and uses a Google Cloud virtual machine with a 4-core CPU and 8GB of memory as the test environment. The client runs on a Google Cloud virtual machine with a 2-core CPU and 4GB of memory, and deploys applications based on Tomcat 7 and Java. The database uses MySQL 5.7 for data storage and management to support data persistence and high-concurrency reading and writing of the task scheduling system. The configuration of the overall test environment ensures that the functional and high-availability tests of the system under the distributed architecture can proceed smoothly and simulate the actual production environment as much as possible.

4.3 Test Data

In terms of test data design, the system tests simple tasks and parallel tasks respectively to verify the scheduling and execution of different task types. In the simple task test, the task ID is 1, the description is "simple task test", the trigger mode is timed trigger, the time expression is "0 0/1 * * * ?" (executed once every minute), and the task custom parameter is TestParam. The task processing implementation logic is: record the task ID, task description and current system time in the log, and return a response that the task is successfully executed. This test is used to verify whether the system can correctly trigger and execute basic tasks, and check the accuracy of task log records.

In the parallel task test, the task ID is 2, and the description is "parallel task test". It also adopts the timed trigger method and uses the same time expression "0 0/1 * * * ?". The task custom parameter is TestParam. The execution logic of the task adopts a two-level processing mode. The first-level task is used to dispatch subtasks, and a total of 100 subtasks (data type is String) are generated; the second-level task is responsible for specific business logic processing, outputs the received subtasks in the log, and returns a successful execution response. The entire test is divided into two stages: integration testing and high availability testing. First, integration testing is carried out to ensure that each subsystem can complete the task collaboratively; finally, special high availability testing is carried out on each subsystem to simulate different failure scenarios to verify the stability and fault tolerance of the task scheduling system.

4.4 Test Results

First of all, it mainly focuses on simple task trigger execution tests, parallel computing task trigger execution tests, console task trigger execution tests, etc. in an integrated test environment. The test results are shown in Table 2-3.

Table 1: Integration Testing 1

Use Case Title	Simple task triggers execution test
Prerequisites	Start the server (core scheduling subsystem), start the client (scheduling execution subsystem), start the console (scheduling control service subsystem)
Test steps	1. Create a test task group and a test task through the console page 2. The client implements a simple task interface and writes the test business processing logic 3. The client configures the test task group ID and starts 4. The client receives the scheduled task schedule according to the time expression configured in the test task and executes the test business logic
Expected Results	The Client receives the scheduled task schedule according to the time expression configured in the test task and executes the test business logic.
Test Results	Meets expectations

Table 2: Integration Testing 2

Use Case Title	Console task trigger test
Prerequisites	Start the server (core scheduling subsystem), start the client (scheduling execution subsystem), start the console (scheduling control service subsystem)
Test steps	1. Create a test task group and a test task through the console page (set the trigger type to API trigger) 2. The client implements the parallel task interface and writes the test business processing logic 3. The client configures the test task group ID and

	starts 4. Click Trigger once on the console page, the client receives the scheduled task schedule and executes the test business logic
Expected Results	The console page is clicked once, the Client receives the scheduled task schedule and executes the test business logic
Test Results	Meets expectations

Table 3: Integration Testing 3

Use Case Title	Parallel computing tasks trigger execution tests
Prerequisites	Start the server (core scheduling subsystem), start the client (scheduling execution subsystem), start the console (scheduling control service subsystem)
Test steps	1. Create a test task group and a test task through the console page 2. The client implements the parallel task interface and writes the test business processing logic 3. The client configures the test task group ID and starts 4. The client receives the scheduled task schedule according to the time expression configured in the test task
Expected Results	The Client receives the scheduled task schedule according to the time expression configured in the test task and executes the test business logic
Test Results	Meets expectations

Secondly, this paper conducted a high availability test. The experimental results are shown in Table 4-6. The goal of high availability testing is to ensure that the system has an overall availability of 99.99% when there is no single point of failure in the core components (Server, Console, ZooKeeper, Client). In the actual production operation process, the system has not had any abnormal task scheduling caused by hardware or software failures, which verifies the stability of its high availability design. During the test process, the high availability test was focused on the Server component and the Console component. The designed test cases included simulating the failure of some servers (1 or 2) in the cluster to observe whether the system can still operate normally and ensure that the task trigger scheduling service is not affected, thereby verifying the system's fault tolerance and the effectiveness of the high availability mechanism.

Table 4: High availability test results 1

Use Case Title	High availability test of core scheduling subsystem
Prerequisites	Start 3 servers (core scheduling subsystems), start the client (scheduling execution subsystem), and start the console (scheduling control service subsystem)
Test steps	1. Start the subsystems of the distributed timed task scheduling system, including 3 servers 2. When the timed task starts periodic scheduling, randomly stop 2 of the servers 3. The task can still be triggered normally on the client
Expected Results	When a server in the server cluster crashes, the core scheduling subsystem can still work and trigger tasks normally.
Test Results	Meets expectations

Table 5: High availability test results 2

Use Case Title	API service subsystem high availability test
Prerequisites	Start the server (core scheduling subsystem), start the client (scheduling execution subsystem), start the console (scheduling control service subsystem)
Test steps	1. Start each subsystem of the distributed timed task scheduling system, including 3 Consoles 2. Configure nginx in the Console cluster to balance, and randomly stop 2 of the Consoles 3. SDK requests can be processed normally
Expected Results	When a machine in the Console cluster crashes, the API service subsystem can still work and process SDK requests normally.
Test Results	Meets expectations

Table 6: High availability test results 3

Use Case Title	High availability test of scheduling control service subsystem
Prerequisites	Start the server (core scheduling subsystem), start the client (scheduling execution subsystem), start the console (scheduling control service subsystem)
Test steps	1. Start each subsystem of the distributed timed task scheduling system, including 3 consoles 2. Configure nginx in the console cluster to balance, and randomly stop 2 of the consoles 3. Tasks can still be triggered normally on the client, and the console page can be used normally
Expected Results	When a machine in the Console cluster crashes, the scheduling and control service subsystem can still work and trigger tasks normally.
Test Results	Meets expectations

5. Conclusion

This study explores the design and implementation of a high-availability distributed scheduled task scheduling system, addressing the challenges of traditional single-machine scheduling methods, such as single points of failure, concurrency bottlenecks, and unstable task execution. By leveraging a multi-machine hot-standby mechanism, a lock-based task trigger strategy, and ZooKeeper for task coordination, the proposed system ensures reliable task execution while minimizing redundant execution. Additionally, dynamic load balancing and asynchronous RPC communication enhance scheduling efficiency and scalability. Experimental results validate the system's robustness, demonstrating its ability to maintain stable task execution even under multi-node failures, making it suitable for enterprise applications in industries such as finance, telecommunications, and the Internet.

Despite these advancements, further optimization remains necessary to enhance system adaptability and intelligence. Future research can explore machine learning-based task scheduling strategies, enabling dynamic resource allocation based on real-time workload patterns. Additionally, incorporating fault prediction mechanisms could proactively identify and mitigate system failures before they occur, further improving system reliability. Furthermore, integrating more fine-grained monitoring and anomaly detection capabilities will provide real-time insights into task execution performance, enhancing system observability and maintainability.

Looking ahead, as distributed computing continues to evolve, the proposed scheduling system can be extended to support edge computing and cloud-native environments. Future iterations may integrate with container orchestration platforms

like Kubernetes to achieve more flexible and efficient scheduling across heterogeneous infrastructures. Moreover, leveraging blockchain technology for task verification and execution tracking could enhance security and transparency in large-scale distributed scheduling. By continuously refining scheduling algorithms and system architecture, this research lays the foundation for the next generation of intelligent, autonomous, and highly scalable task scheduling solutions.

References

[1] Perera C. Optimizing Performance in Parallel and Distributed Computing Systems for Large-Scale Applications[J]. Journal of Advanced Computing Systems, 2024, 4(9): 35-44.

[2] Zangana H M, khalid Mohammed A, Zeebaree S R M. Systematic review of decentralized and collaborative computing models in cloud architectures for distributed edge computing[J]. Sistemasi: Jurnal Sistem Informasi, 2024, 13(4): 1501-1509.

[3] Manne U K. High-Availability Database Architectures for Autonomous Vehicles: Ensuring Real-Time Performance and Reliability[J]. INTERNATIONAL JOURNAL OF RESEARCH IN COMPUTER APPLICATIONS AND INFORMATION TECHNOLOGY (IJRCAIT), 2024, 7(2): 785-796.

[4] Huang J, Zhang Y, Xu J, et al. Implementation of seamless assistance with Google Assistant leveraging cloud computing[J]. Applied and Computational Engineering, 2024, 64: 168-174.

[5] Y. Wang, "Optimizing Distributed Computing Resources with Federated Learning: Task Scheduling and Communication Efficiency," Journal of Computer Technology and Software, vol. 4, no. 3, 2025.

[6] Y. Deng, "A Reinforcement Learning Approach to Traffic Scheduling in Complex Data Center Topologies," Journal of Computer Technology and Software, vol. 4, no. 3, 2025.

[7] X. Sun, Y. Duan, Y. Deng, F. Guo, G. Cai, and Y. Peng, "Dynamic Operating System Scheduling Using Double DQN: A Reinforcement Learning Approach to Task Optimization," arXiv preprint arXiv:2503.23659, 2025.

[8] Q. He, C. Liu, J. Zhan, W. Huang, and R. Hao, "State-Aware IoT Scheduling Using Deep Q-Networks and Edge-Based Coordination," arXiv preprint arXiv:2504.15577, 2025.

[9] Y. Ren, M. Wei, H. Xin, T. Yang, and Y. Qi, "Distributed Network Traffic Scheduling via Trust-Constrained Policy Learning Mechanisms," Transactions on Computational and Scientific Methods, vol. 5, no. 4, 2024.

[10] X. Li, Y. Peng, X. Sun, Y. Duan, Z. Fang, and T. Tang, "Unsupervised Detection of Fraudulent Transactions in E-commerce Using Contrastive Learning," arXiv preprint arXiv:2503.18841, 2025.

[11] D. Xu, "Transformer-Based Structural Anomaly Detection for Video File Integrity Assessment," Transactions on Computational and Scientific Methods, vol. 5, no. 4, 2024.

[12] A. Liang, "A Graph Attention-Based Recommendation Framework for Sparse User-Item Interactions," Journal of Computer Science and Software Applications, vol. 5, no. 4, 2025.

[13] F. Guo, X. Wu, L. Zhang, H. Liu, and A. Kai, "A Self-Supervised Vision Transformer Approach for Dermatological Image Analysis," Journal of Computer Science and Software Applications, vol. 5, no. 4, 2025.

[14] L. Zhu, "Deep Learning for Cross-Domain Recommendation with Spatial-Channel Attention," Journal of Computer Science and Software Applications, vol. 5, no. 4, 2025.

[15] M. Li, R. Hao, S. Shi, Z. Yu, Q. He, and J. Zhan, "A CNN-Transformer Approach for Image-Text Multimodal Classification with Cross-Modal Feature Fusion," 2025.

[16] Y. Zhang, "Social Network User Profiling for Anomaly Detection Based on Graph Neural Networks," arXiv preprint arXiv:2503.19380, 2025.

[17] Q. Sun and S. Duan, "User Intent Prediction and Response in Human-Computer Interaction via BiLSTM," Journal of Computer Science and Software Applications, vol. 5, no. 3, 2025.