ALLUXIO

Thousand-Node Alluxio Cluster Powers Game AI Platform Architecture Evolution for Interactive Queries

A Production Case Study from Tencent

Bing Zheng, *Software Engineer*
Baolong Mao, *Senior Software Engineer*
Zhizheng Pan, *Senior Software Engineer*

What's Inside

CASE STUDY

Tencent is one of the largest technology companies in the world and a leader in the gaming sector. The game AI platform supports AI research and development at Tencent. To provide model training with the best experience, Tencent has implemented a 1000-node Alluxio cluster and designed a scalable, robust, and performant architecture to accelerate the game AI training.

In this case study, you will learn:

- What are the data challenges in model training at Tencent
- Why does Tencent choose Alluxio
- How to design an architecture that suits large-scale Alluxio deployment

1 / Overview

Alluxio is an open-source data orchestration platform for AI and data analytics applications. With an increasing number of data analytics and AI applications being containerized, Alluxio is becoming the top choice for large organizations as an intermediate layer to accelerate analytics and model training.

We have adopted Alluxio to solve the data challenges in distributed computing for our game AI offline training. Traditionally, this is solved by:

- Image packaging: Although resource isolation is better in this way, the capacity of the image caching is limited. Frequent data updates result in repackaging and re-deploying the image, forcing the container services to stop each time.
- Load on-demand: Although this architecture is simple, we have to pull data remotely, which results in poor performance.
- Deploy locally: This solution brings good data locality. However, authentication and DevOps are challenging.

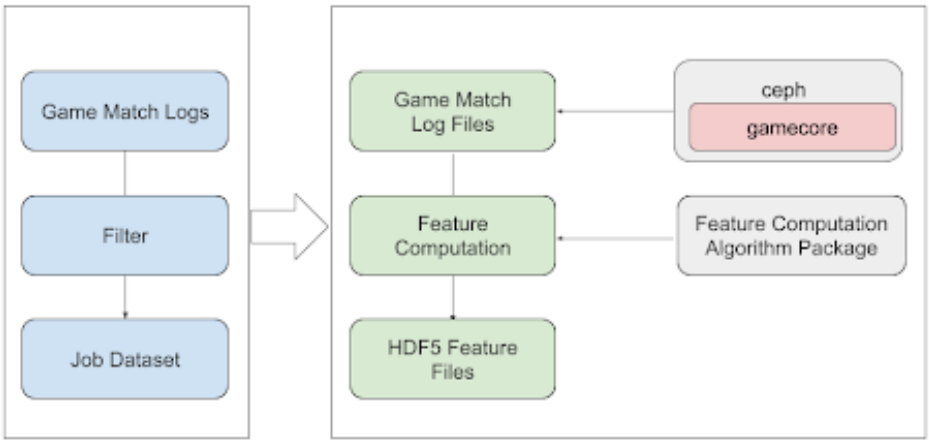
These solutions have their pros and cons. With Alluxio, we have solved the challenges in a better way. The implementation of Alluxio can significantly improve the concurrency limit of AI workloads without adding additional costs. Also, the business side is not impacted by the change of storage system because AI workloads are still using the original POSIX interface to access Alluxio.

In this case study, we discuss the implementation and optimization of Alluxio for our game AI platform.

2 / Business Background and Data Challenges

The game AI offline training consists of supervised learning and reinforcement learning. Generally, supervised learning is divided into three stages: feature computation (feature extraction), model training, and model evaluation. Reinforcement learning requires feature computation as well. In the game AI offline training, feature computation requires reducing game match information to generate the feature data for model training through statistics and computation. The information recovery of the game match is made through the corresponding game dependencies (game ontology, game translator, game replay tool, etc.), which generally range from 100MB to 3GB in file size. Furthermore, due to the versioning nature, processing specific game match information requires a specific version of the game dependency.

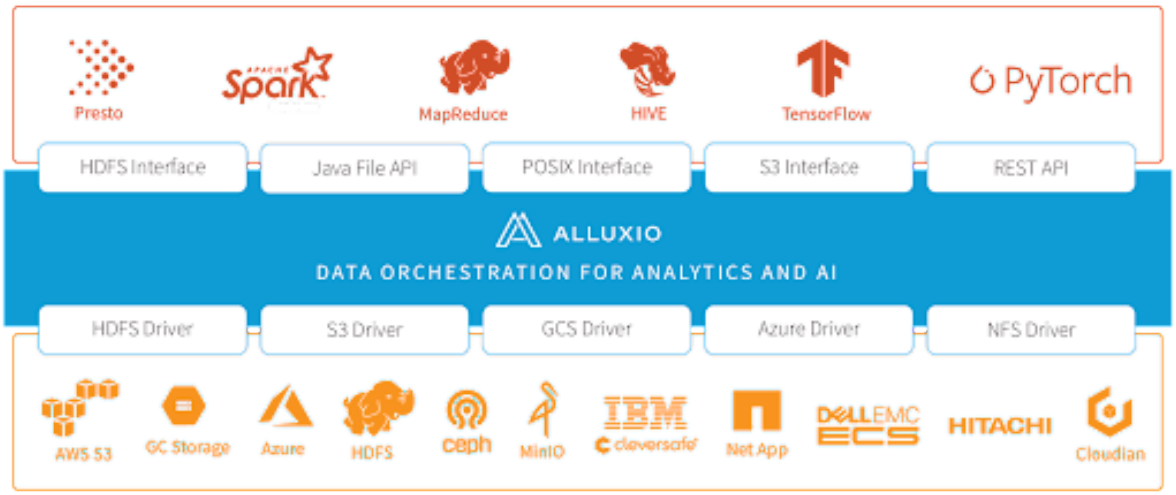
The game dependency on the storage side is called gamecore, which corresponds to a Linux client of a specific game version. Gamecore can be stored on local storage for better performance and stability, but it is expensive and requires authentication of local machines.



Alternatively, gamecore can be stored in distributed storage, such as Ceph, which is faster and easier to deploy. Still, the disadvantage is that the MDS, metadata server daemon of cephfs, will become a bottleneck. Usually, thousands of containers are scheduled in feature computing jobs. Each container will start several processes. At the beginning of the job, tens of thousands of processes will access the same gamecore in parallel with hundreds of gigabytes of data reads per minute on the storage side. The metadata pressure will fall solely on MDS because these files are small. In addition, the latency between storage and compute is usually high, especially when they are remote, which may lead to a higher job failure rate.

With careful consideration, we decided to introduce Alluxio on Ceph in order to address the current data challenges. In this implementation, we got strong support from the game AI team and the operation team. The game AI team gave us the overall business background and coordinated the production environments so that we could fully test the solution before productionalization. The Operations Management team also provided significant assistance with deployment architecture support and resource coordination.

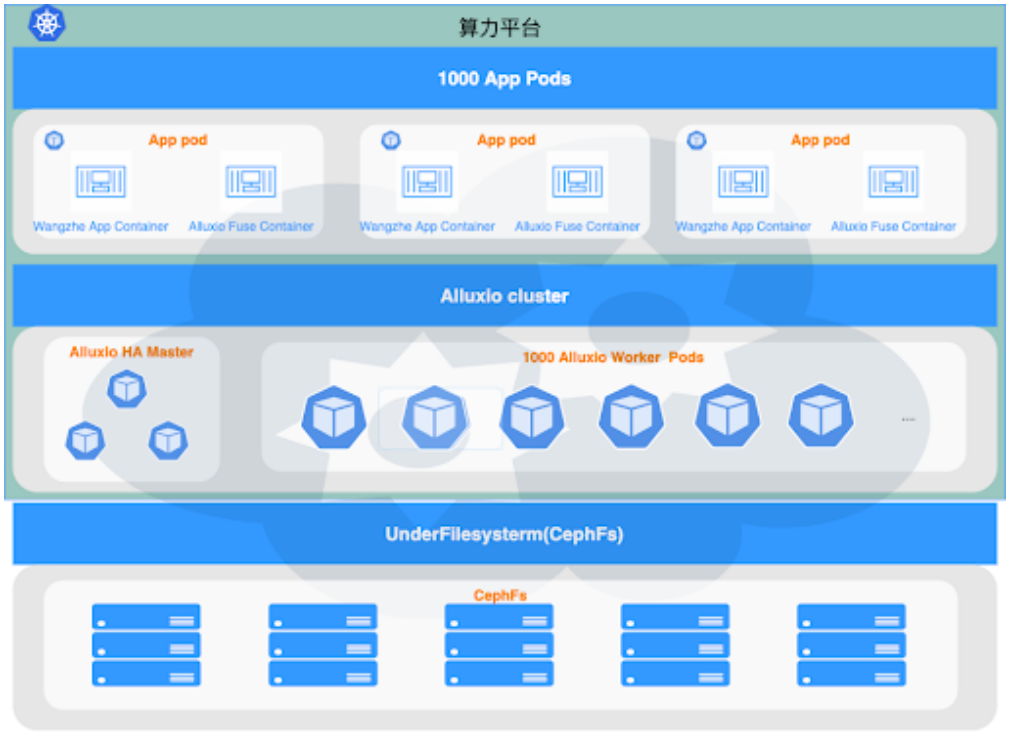
3 / Architecture with Alluxio



In the data analytics ecosystem, Alluxio sits between the data-driven compute framework and the storage systems. Alluxio unifies the data stored in these different storage systems, providing flexible APIs and a global namespace for data-driven applications. In our use case, the under storage is cephfs, and the data application is feature computing. Alluxio, as an intermediate layer, is well suited as a distributed caching solution that is ideal for optimizing reads-heavy, massive small files, and high concurrent access scenarios of feature computing business. We have gained several benefits from Alluxio:

- Alluxio provides good support on the cloud, making it easy to deploy and scale up and down Alluxio clusters on computing platforms.
- Alluxio can be deployed co-located with the computation side. By deploying Alluxio workers on the same node of the compute, we can cache data locally and improve I/O throughput.
- The Alluxio worker uses the memory disk of the computing node, which can provide sufficient cache space, and loads hot data from cephfs into the worker through distributedLoad.

The following diagram shows the architecture of Alluxio in the game AI platform. In production, we want to support concurrent jobs on 4000 CPU cores to run stably. We configure quad-core CPU for each pod of game match feature extraction, providing 1000 pods concurrency on the application side. Each pod embeds an alluxio-fuse sidecar container as a client, and data read requests from the application can directly access data from Alluxio through POSIX with the path mounted by alluxio-fuse.



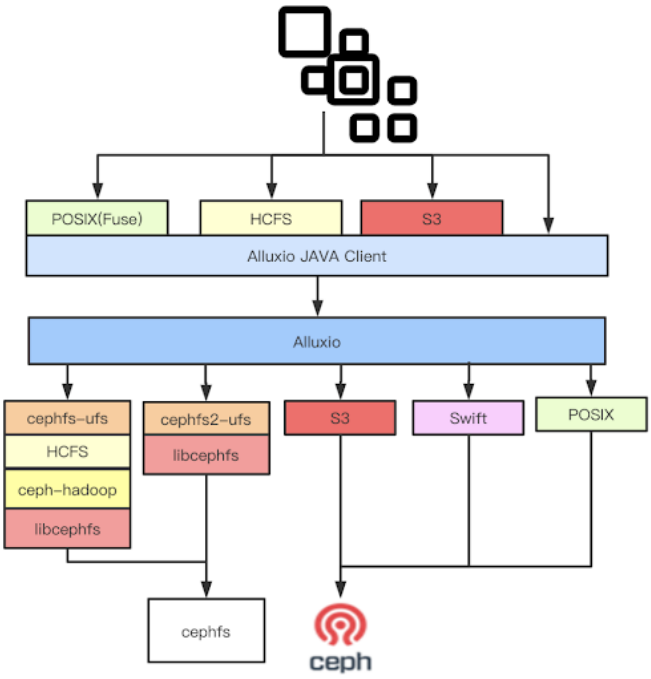
The Alluixo cluster master node is configured in HA mode, and the worker number is 1000. We want to co-locate the application pods and worker pods to one node as much as possible so that we can use the domain socket to improve the read performance further. Before the application goes live, the hot data of gamecore in cephfs is preloaded to the Alluxio worker through distributedLoad to warm up

4 / R&D and Performance Tuning

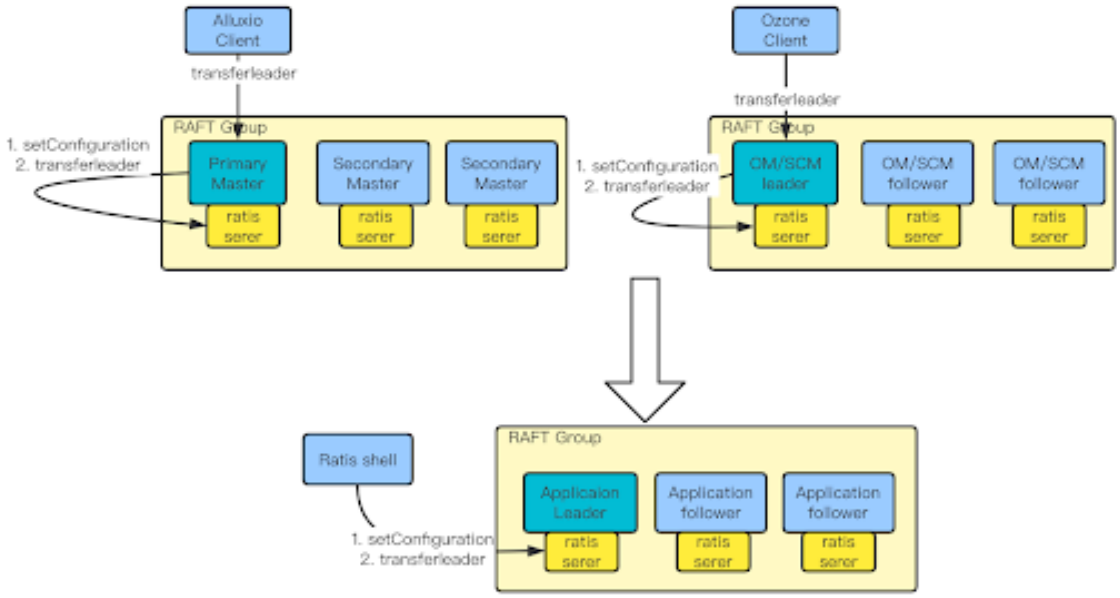
The Alluxio cluster is currently supporting feature computing and is a large-scale deployment (1000+ worker nodes) in AI and machine learning platform. Such extensive concurrent access poses challenges to the master's resilience. We have done tuning and added new features to achieve the best results during our practice.

R&D

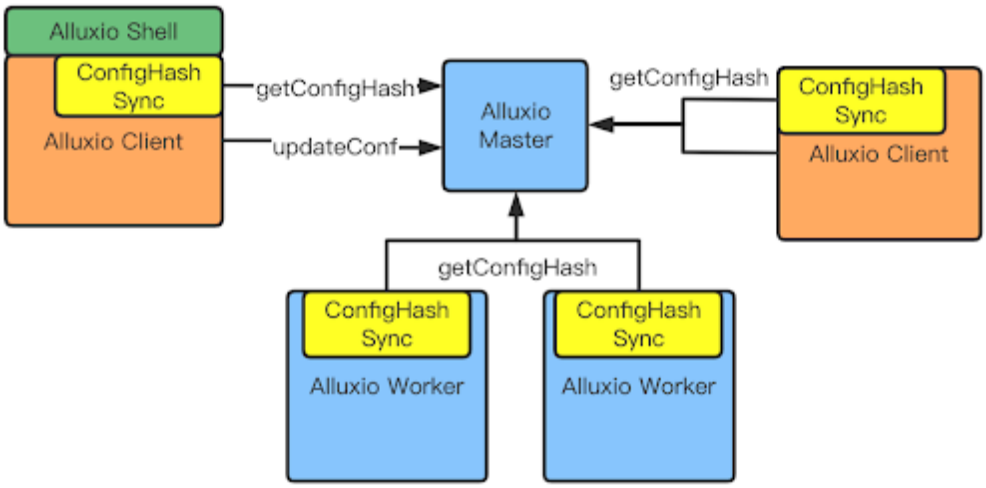
- Alluxio provides good support on the cloud, making it easy to deploy and scale up and down Alluxio clusters on computing platforms.



- We worked with the Alluxio community to design and implement the feature of switching leader to a specified master node for HA.
    - We abstracted the ratis-related functionality into a separate ratis-shell tool (https://github.com/opendataio/ratis-shell), which can set the priority of each master by sending setConfiguration requests to the ratis server directly, followed by sending transferleader request to ensure that the leader has switched to the specific node we want. Ratis-shell works with Alluxio and Ozone, and all other applications that use ratis.



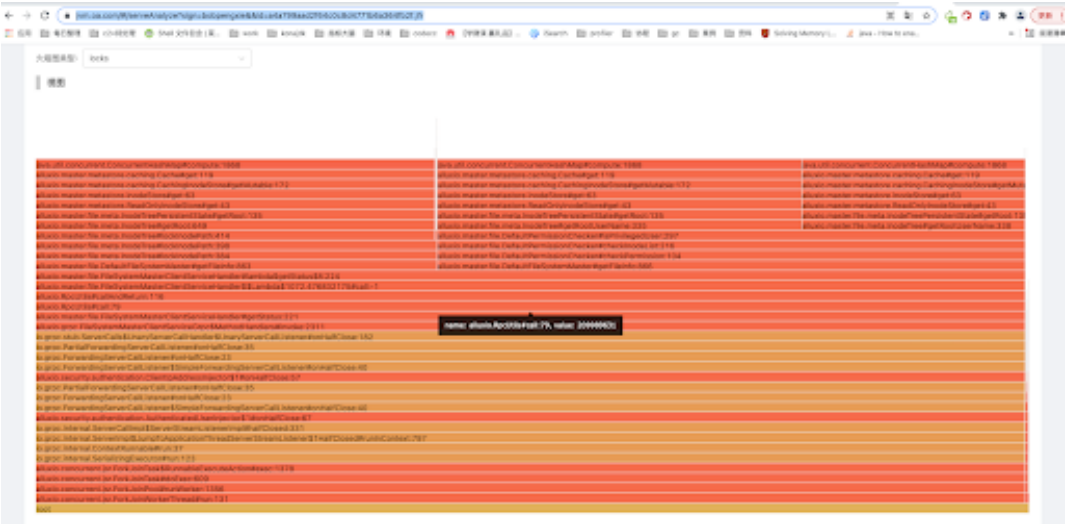
- We added the ability to dynamically change the configuration to modify specific cluster parameters online to optimize the configuration with minimal impact on the application.
    - As shown below, the updateConf API was added between the Client and the Master. Through the API, the configuration change requests can be sent to the Alluxio Master, and the internal config hash will be changed after the master updates the configuration. In addition, other services connected to Alluxio Master, such as Client and worker, will periodically synchronize their config hash with the master and perceive the configuration changes and thus synchronize the configuration changes.



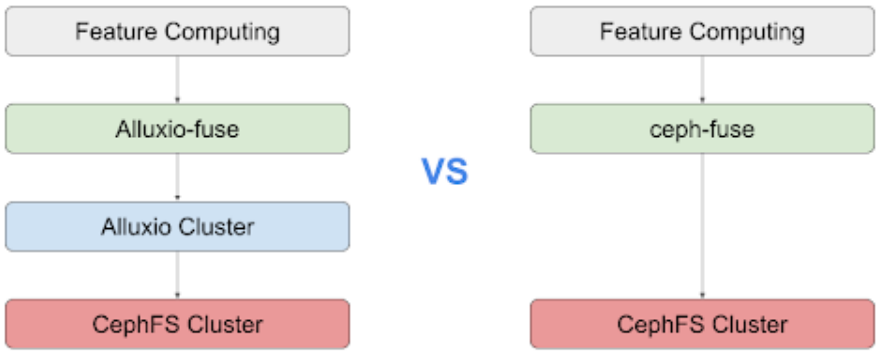
- On the Alluxio FUSE client side, for the all-read scenario, we have turned on the kernel cache and the metadata cache on the Alluxio client side to further improve the read performance. We also optimized the metadata cache. For example, when the metadata of the under storage changes, we can actively invalidate the metadata cache and recache it, which is very flexible. We also fixed some bugs after Alluxio FUSE turns on the LocalCache.
- We added metrics of master busy status, ratis metrics, OS, JVM, GC, cache hit rate, and many other valuable metrics to enrich Alluxio's metrics system.
- We developed a feature to view stacks of Alluxio's key processes to track cluster status.
- We leveraged the warm-up feature of distributedLoad to fix an OOM issue in the Alluxio Job Service when executing many distributedLoads.

Configuration Optimization

- Turn off passive cache. By default, Alluxio turns on the passive cache feature Allluxio.user.file.passive.cache.enabled=true, so if the client finds that the data block is not in the local worker, it will copy from the remote worker to the local worker, resulting in many copies of each worker. This configuration adds significant metadata pressure to the Alluxio master in our scenario with 1000 workers. Our tests have shown that the performance benefit of this locality is very small, so we turned it off to reduce the pressure on the master.
- Use internal kona jdk11 and tune the jvm parameters for master and worker. By using konajdk11 + g1gc, there were no longer leader master switching issues due to fullgc.
- Turn off auditlogging. With the help of the jvm team, we identified that the enabled auditlog is the bottleneck of the all-read scenario. By setting alluxio.master.audit.logging.enabled=false to turn off auditlogging, the throughput improved by 7x. The flame chart captured by kona-profiler reveals that using the ROCKSDB metadata management has increased the performance overhead, and we plan to switch Alluxio to HEAP metadata management in the future.



5 / Benchmarking Test



We use feature computing jobs of a moba (multi-player online battle arena) game to compare Alluxio (UFS is cephfs) and cephfs, respectively. The cluster information of Alluxio in the test is as follows:

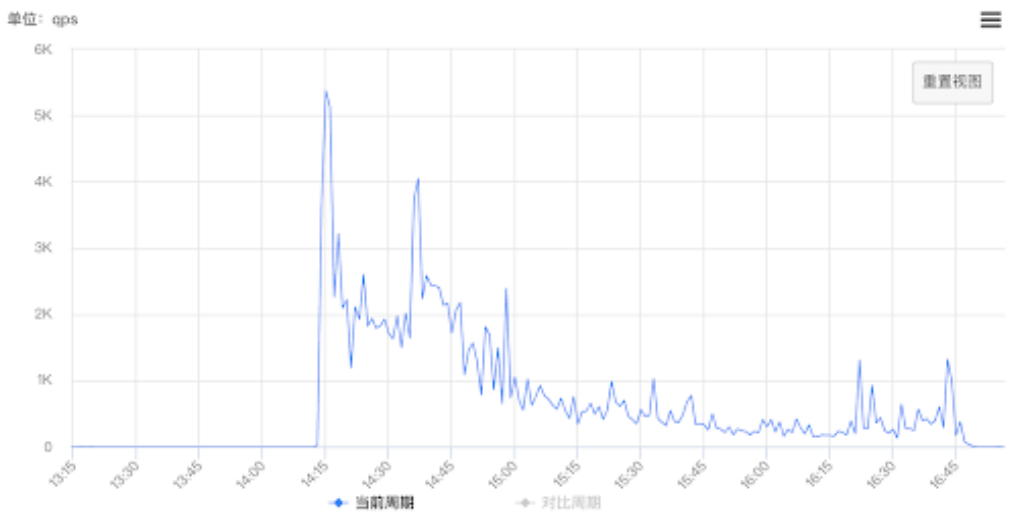
- Alluxio master: 3 masters deployed in High Availability mode.
- Alluxio worker: 1000 workers, about 4TB of storage space.
- Application pods: 1000, each pod is in parallel with 4 cores.
- Benchmarking test: A moba game AI feature computing job (containing 250,000 matches).

Test results are as follows:

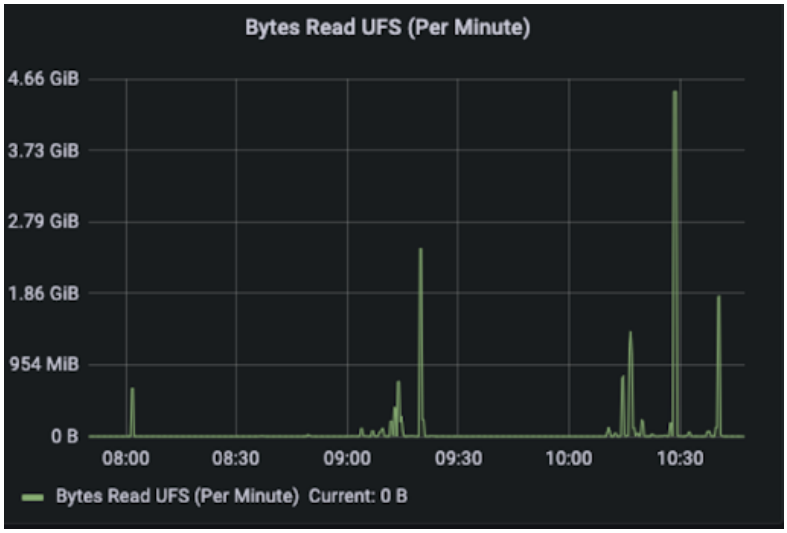
Client	Jobs	Completion	Failure	Time	Failure rate
alluxio-fuse	250000	248152	1848	2h46min	0.73%
ceph-fuse	250000	242930	7070	2h40min	2.8%

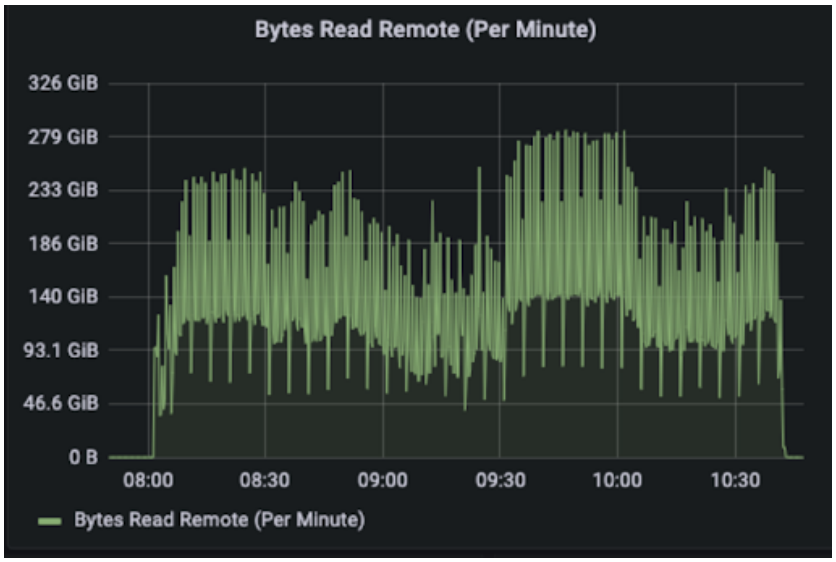
- Accessing Alluxio + cephfs has a failure rate of 0.73% and a job duration of 2h 46min.
- Accessing cephfs directly has a failure rate of 2.8% and a job duration of 2h 40min.

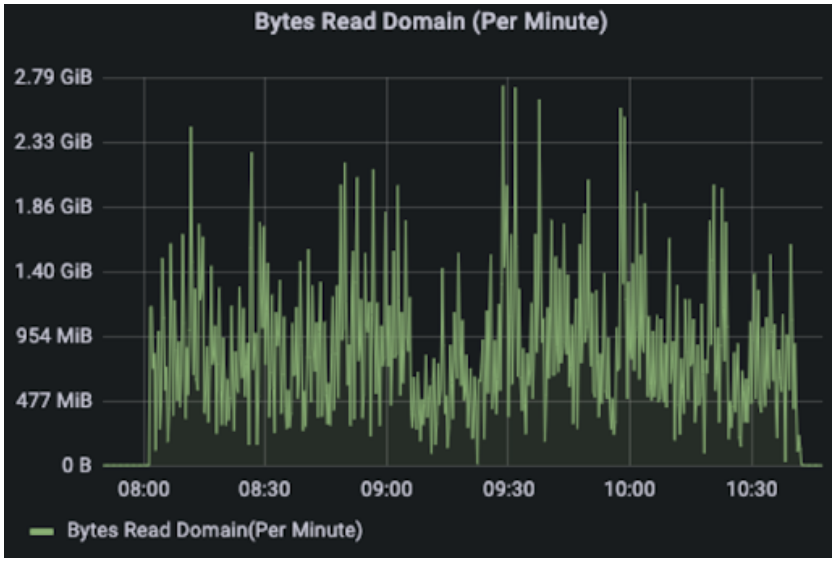
Both solutions pass the tests above and the failure rate is within an acceptable range. With Alluxio + cephfs, failure rates are lower.



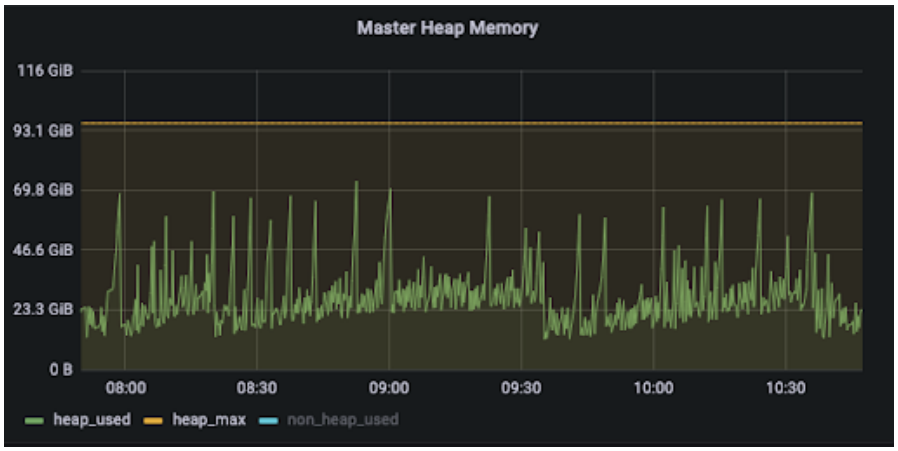
The graph above shows the metadata stress metrics (rpc count and qps of mds) for Alluxio and cephfs. There is a pressure hit at the beginning of the job, and then the pressure of master metadata gradually decreases. With Alluxio, the qps of ceph mds is almost zero, which indicates that Alluxio is reducing most of the pressure.







The Read Remote, Read UFS, and Read Domain metrics are used to observe the locality of the data. We can see that Remote Read and Read Domain account for most of the read traffic, with most of the read traffic being remote reads between workers and local domain socket reads, and very few from UFS.



The above figure shows the heap memory change curve during job execution after using kona jdk11. While using the official version of kona jdk11 before, the master encountered a leader switching problem due to the long gc time. After replacing kona jdk11, this no longer happened, and master has been smoother.

6 / Future Work

Increase Throughput Limit

Currently, Alluxio master is under 7000 cores concurrency pressure, and we have found that the callqueue of master is backlogged, and the masterStressBench tool has measured the throughput capacity of nearly 210,000 rpc requests per second.

To make the alluxio cluster support endless concurrent access, tuning a single cluster is not enough. We need to design the overall architecture to support higher concurrent access.

Decouple Application and Alluxio FUSE with Alluxio CSI

Currently, Alluxio-FUSE is located in the same pod as the application as a sidecar, so the application container and Alluxio-FUSE container must be managed together. By decoupling Application and Alluxio FUSE with Alluxio CSI, we can independently maintain the pod and yaml on the application side.

Build the Alluxio Cluster Management System on Kubernetes

We maintain a helm chart for operating and maintaining Alluxio clusters based on the helm chart template provided by Alluxio. Still, we want to go further and operate each pod and container based on the Kubernetes API and interactively execute the commands such as the mount and unmount of under storage, job service visualization, and load free services.

7 / Summary

With Alluxio, we have been able to support 4000 concurrent CPU cores smoothly for the game AI feature extraction use case. From our practice, Alluxio has proven to offset most of the metadata pressure for the under storage and reduce the failure rate of jobs to a satisfactory range. Such large-scale, high-concurrency scenarios posed challenges to Alluxio, and we have contributed to the open-source community and improved the stability, usability, and DevOps ability of Alluxio, making it more adaptable in the future.

8 / Reference

- Kaiwu Game AI Offline Experiment Platform
- ratis-shell
- Alluxio Documentation
- Chinese Version of This Case Study

About the Authors

Bing Zheng

- Software Engineer at Tencent
- Responsible for the R&D of big data system and distributed storage.

Baolong Mao

- Senior Software Engineer at Tencent, Alluxio PMC member, Apache Ozone committer, Alluxio OTeam open source collaborative team leader at Tencent
- Responsible for the development and implementation of Alluxio at Tencent, and the development of the Apache Ozone file system.

Zhizheng Pan

- Senior Software Engineer in Kaiwu game AI platform team at Tencent
- Responsible for developing and operating the machine learning platform for game AI use cases.