ALLUXIO

Speeding Up the Atlas Supercomputing Platform with Fluid + Alluxio

Dongdong Lv *(Platform Architect at Unisound)*
Qingsong Liu *(Algorithm Researcher at Unisound)*

What's Inside

CASE STUDY

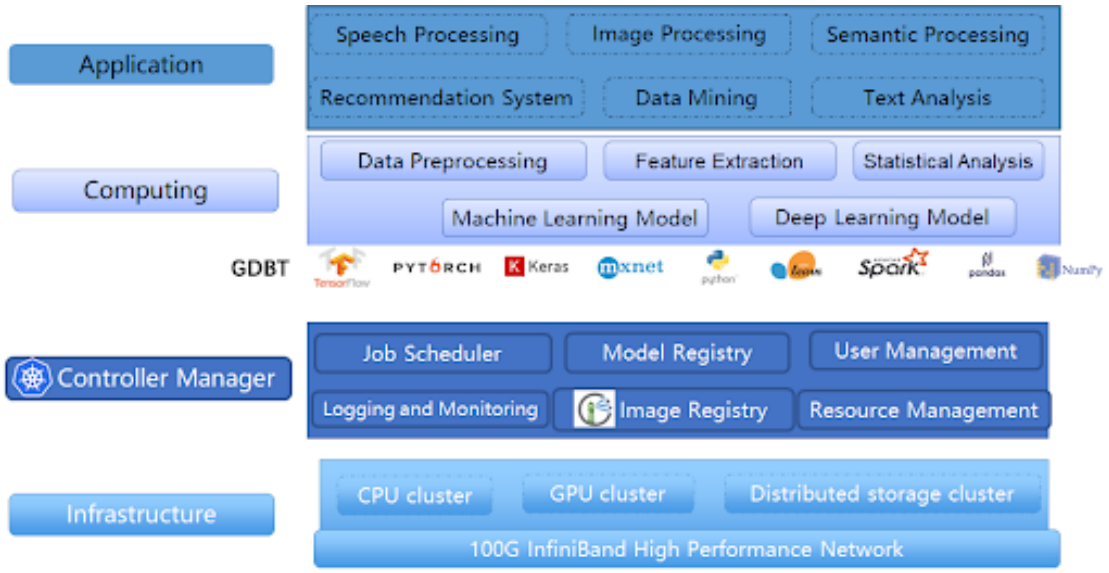
1 / Introduction to Unisound and Atlas AI Platform

Unisound is an artificial intelligence company focusing on Internet of Things services. Unisound's AI technology stacks include the perception and expression capabilities of signals, voices, images, and texts, and the cognitive technologies such as knowledge, understanding, analysis, and decision-making, towards a multi-modal AI system. Atlas is the supercomputing platform supporting all kinds of AI applications including model training and reasoning inferencing.

Unisound has built the industry-leading GPU/CPU heterogeneous computing and distributed file system, called Atlas. This platform provides AI applications with high-performance computing and data access capabilities at a massive scale. Based on the Kubernetes open source architecture, the Unisound team has developed the core features and successfully built an AI supercomputing platform with a floating-point processing capacity of more than 10 PFLOPS (100 million times per second). The platform supports the main machine learning frameworks, and developers can efficiently research and develop core applications such as voice, NLP, big data, multimodal, etc. The platform also serves external customers such as SMBs and research institutions with customized computing and storage capabilities.

2 / Problems and Challenges

On the Atlas platform, computation is decoupled from storage. At present, the interconnections among the storage servers, the computing servers, and between the computing and storage servers are 100GB InfiniBand.



We have built our own high-performance distributed file system, named Lustre, as the storage layer of the Atlas platform. Lustre consists of several petabytes of model training datasets. The Lustre distributed file system is compatible with the POSIX interface so that various deep learning frameworks can directly read data from Lustre. The separation between computing and storage enables them to scale independently, making overall architecture more flexible. However, such architecture encountered problems such as slow data access and network bandwidth bottlenecks. The challenges are as follows:

The I/O Bottleneck

As the number of users grows, there is a large increase in the bandwidth, metadata workload, and server workload with unchanged storage resources. In the storage cluster, there are many workloads running on the same single GPU node, competing for I/O resources. Thus, the entire training cycle gets longer, which greatly reduces the efficiency of the research and development.

Massive Small Files

The second challenge is about the training data set itself. In the noise reduction training, certain users would process terabytes of small files, which puts great pressure on the metadata service of the distributed file system, making it inefficient in reading data. The slow read lowers the overall utilization of the GPU, which increases the overall model training time.

Many Data Types

Many applications run on the Atlas platform with different data formats and file sizes, making it nearly impossible to use one single configuration to fit all services. Based on user behaviors, we found that the most used data is for model training, with the rest for model inference and CPU-intensive data generation applications.

Data Redundancy

The last challenge is the dataset replication on the platform. When the same dataset is used by different users in the same group or in different groups, multiple copies are stored, resulting in a waste of storage space.

3 / The Previous Solution

We want to find a solution that is not only cost-effective but also requires the least architectural changes to deal with the I/O bottleneck and offload the metadata server. Below are several ways we explored.

Put Limitation on Bandwidth

Massive concurrent reads will push the bandwidth to reach the limit, causing storage system freezes or failures. We limit the bandwidth by putting limitations on the bandwidth of each client on the computing node and the UID/GID of each user. However, this method is not flexible and cannot fully utilize GPU resources. When there are two large I/O training jobs running on the same node, due to the bandwidth limitations, both training jobs are limited on reads. These limitations make GPU less utilized because it does not benefit from reading in parallel. We found that the utilization rate of GPU in this scenario is only 40%, which means hardware resources are wasted.

Aggregate Small Files to Large Files

We also took steps to deal with pressure put on the metadata by the massive number of small files. We estimated the number of small files by collecting the number inode of and the total storage size of each user to limit the quantity of the small files. Then, we implemented a series of data aggregation tools, allowing users to aggregate small files into large file formats such as lmdb and tfrecord.

Customize the Task Scheduler

To avoid too many jobs running on the same node at the same time, we customized the plug-ins of the task scheduler by adding scheduling policies that identify the resource usage of the computing node so that it can schedule jobs to idle nodes to avoid the I/O competition when running multiple jobs on the same node. However, this solution doesn't work when all the computing nodes are overloaded because competition is inevitable.

Tiered Cache

In order to fully utilize the idle hardware resources and reduce the pressure on the storage system, we developed the V1.0 cache solution as a temporary solution. This solution can relieve the storage pressure to a certain extent, but the data management is not automated yet, so it can only be a temporary solution until we find an ultimate solution and build the new architecture.

4 / The New Architecture

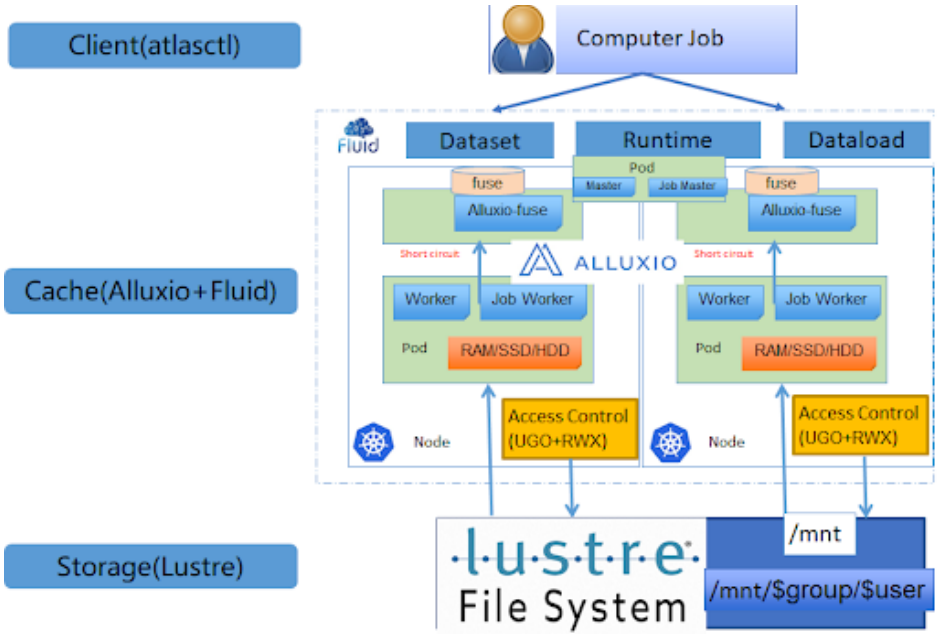
In 2020, the Unisound team started to deploy Alluxio and conducted a series of tests, including functionality tests and performance benchmarking. We found that Alluxio can meet our needs and solve our pain points effectively at a lower cost.

- Alluxio Fuse provides a POSIX file system interface, which allows users to seamlessly use the distributed cache without changing the applications.
- Alluxio supports different types of storage systems, including distributed file systems, object storage, etc. When we introduce new storage to our platform, Alluxio supports it very well, keeping our entire cache architecture stable.
- Alluxio provides intelligent cache management. Alluxio's tiered cache fully makes use of memory, SDD, or HDD, reducing the cost of data-driven applications with elastic scalability.
- Alluxio supports Kubernetes or containers deployment, which is consistent with our existing technology stack;
- Alluxio provides HA support to ensure the high availability of the distributed cache system

Our previous architecture completely decoupled computing and storage. By introducing Alluxio as a cache layer between computing and storage, users can enjoy fast data access because Alluxio brings the underlying storage to memory or local hard drives on each computing node. The entire platform utilizes the resources of both the distributed file system and the local hard disk.

When deploying Alluxio to production, we encountered problems such as access control and data mounting. Fluid provides a more cloud-native way to use Alluxio and manage data. Just like managing cloud resources, Kubernetes schedules and allocates cached data, which is better than the previous V1.0 cache.

Our ultimate architecture is: Alluxio is responsible for data migration and cache management, moving data from the underlying distributed file system to the local cache of the computing node, providing acceleration to the platform applications; Fluid is responsible for the orchestration of caching and applications. Based on Fluid, the platform can perceive caching and intelligently perform cache management without manual operations.



After implementing the new architecture, we integrated Fluid with our self-developed model training task submission tool, atlasctl, to hide the complexity on the user side as much as possible. Users can create a cache dataset by using `atlasctl cache create` and specifying parameters such as cache size and cache media, and so on. This tool allows users to pay more attention to the data and the application itself without having to understand the underlying cache infrastructure.

5 / Configurations Settings

We introduced Fluid + Alluxio as a new architecture to our platform. During the deployment, we encountered some problems in different scenarios. We immediately reached out to the community, and the community solved our needs in a timely manner. Here we would like to talk about several main features.

`hostpath` and `nonroot` Support

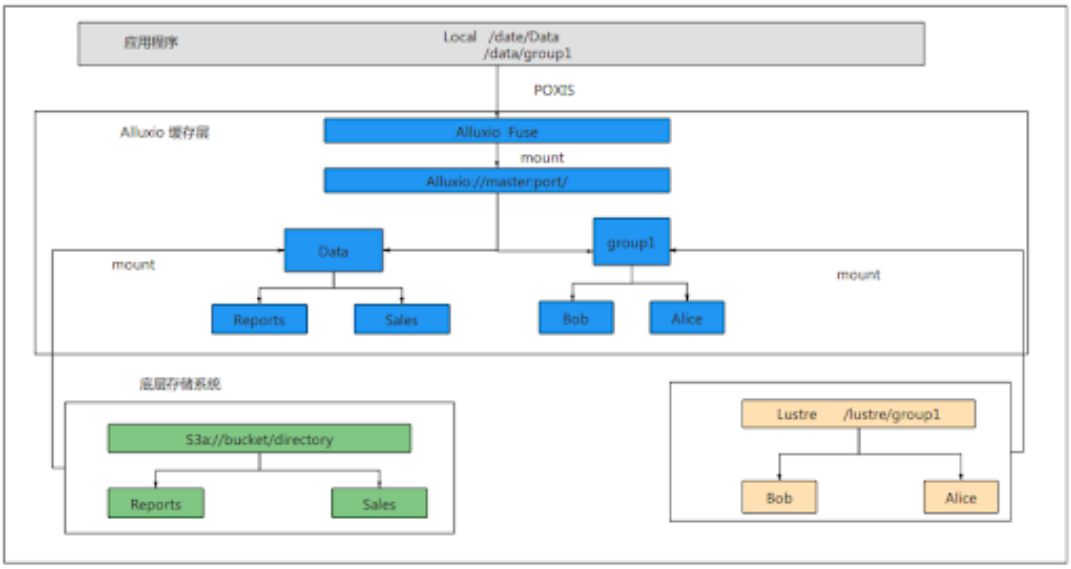
In the Atlas platform, we set `nonroot` for the distributed file system, which means, by default, no root user from the client side would have the permissions to manage the user directory. However, we need Alluxio to access UFS as a root user. We solved this problem using Fluid. If the root user in Allluxio is accessing the UFS, we can use Fluid to set UID and GID respectively on the cache and data side. If we use the same UID and GID settings on the data side, the data can be read from the client side. If the UID and GID of a dataset are set to another user, the data can be shared. The user information on the cache side ensures that Allluxio can successfully read the data in UFS. The feature solves the problems of access control and data redundancy.

Multiple Mount Points Support

The training jobs usually span different data sets, which may be under different directories of the same storage system or different storage systems. Alluxio can provide a unified namespace for applications. Through the abstraction of the unified namespace, applications can access multiple different storage systems through the unified namespace and interfaces. Instead of connecting each storage system to each application, we only need to connect to Alluxio. Through Alluxio Fuse, users can use the POSIX interface to access the data stored from different underlying storage systems.

The unified namespace ensures that the namespace of Alluxio and the underlying storage system remain consistent. The directories and file names of different underlying storage can be mapped in Alluxio.

By using this feature, users can cache and accelerate the data access from two storage systems in the same training jobs at the same time. There is no need for data migration, during which the combination of compression, packaging, migration, and decompression of terabytes of small files would take several hours. Using this feature, users only need to change the storage path of the data for the next job and there is no need to change the source code.



Cache Warmup

The computing resources in the platform are often more scarce than storage resources. When a user initiates a training job using terabytes of small files, the data and metadata sync between the underlying storage system and the cache system will take a long time. Alluxio provides the features of `loadMetadata` and `loaddata`, which are integrated by Fluid. Users can pull the data from the remote storage system to the local distributed cache engine in advance so that applications consuming the data set will get accelerated even if running for the first time. This feature can effectively increase the GPU utilization of the cluster and avoid wasting time when the metadata is synced during the first cache. The application gets a faster I/O at the beginning so that overall GPU utilization increases.

Performance Tuning

Alluxio provides a lot of performance tuning settings. We configured and tuned according to the characteristics of different scenarios. For read-heavy scenarios, we did performance tuning for general settings and for different kinds of data sets.

General settings:

- Open `kernel_cache` and set `alluxio.user.metadata.cache.enabled` to `true` to enable file and directory metadata caching on the client side. For all-read scenarios, configure `alluxio.user.metadata.cache.max.size` and `alluxio.user.metadata.cache.expiration.time` to tune the maximum number of cached metadata cache and expiration time.
- By setting `alluxio.user.file.passive.cache.enabled=false` and `alluxio.user.file.readtype.default=CACHE` to avoid cache jitter brought by frequent eviction (Cache Eviction).

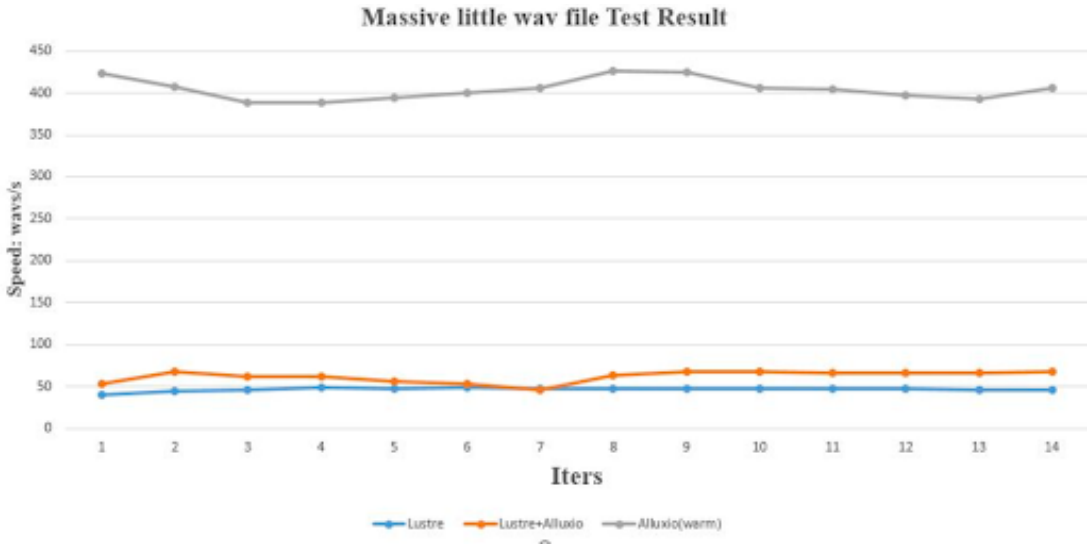
6 / Test Scenarios

We tested three scenarios according to the size of the data set. The first type is small files, with the size of a single file under 1M. The second is medium files in several hundred gigabytes, and the third is terabytes of large files.

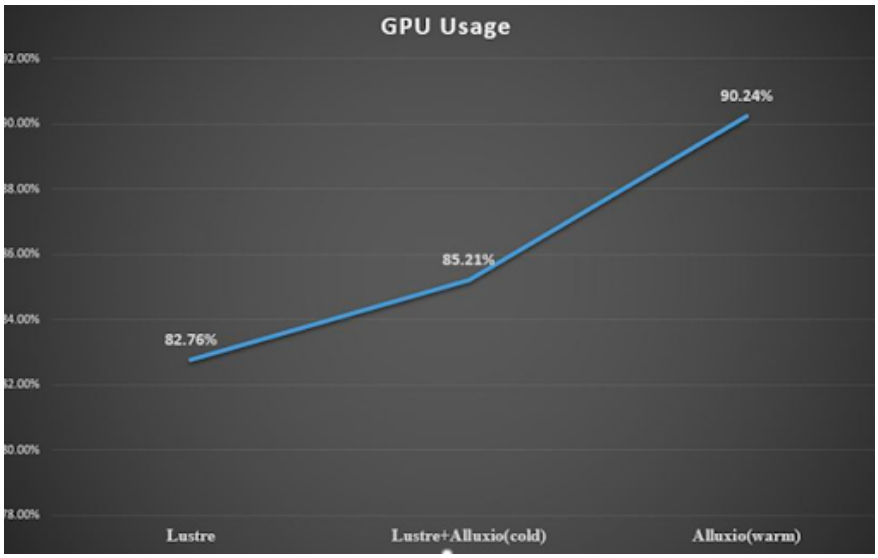
Noise Reduction

This test uses the DLSE model generated by the Pytorch framework. The number of files in the dataset is about 500,000 and the total size is 183 GB. Memory is used as the cache of Alluxio.

We use a single machine with 10 GPU cards for this test. Based on Pytorch's native DDP framework for multi-card communication, we compare the two scenarios: read directly from the distributed file system (Lustre), read from the Alluxio cache (Lustre+Alluxio), and read from Alluxio with cache warmup (Alluxio, warm).



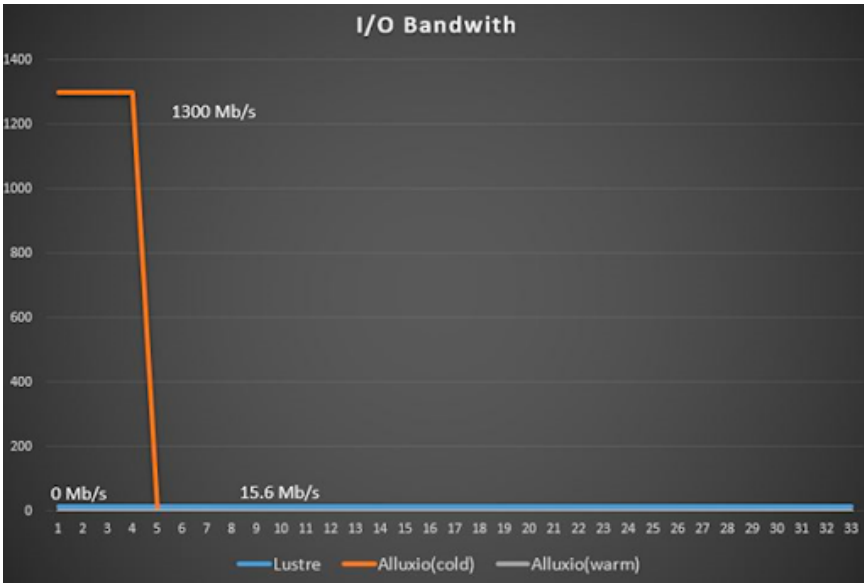
We can see that, during the first read, using Alluxio with cache warmup (Alluxio, warm) is nearly 10x faster compared to reading directly from the distributed file system (Lustre). Because when Alluxio reads the first time, it needs to sync the metadata and cache data at the same time, the real advantage of caching is still not reflected yet. In the second read, since the data have all been brought into the cache, the performance depends on the cache hit rate of Alluxio. From the above test results, we can see that there is a significant speedup.



With a faster data read, we also improved overall GPU utilization. By monitoring the utilization, we found that the GPU usage kept at about 90% using Alluxio cache warmup. At the same time, we can store cached data in the memory, effectively off-loading the underlying storage.

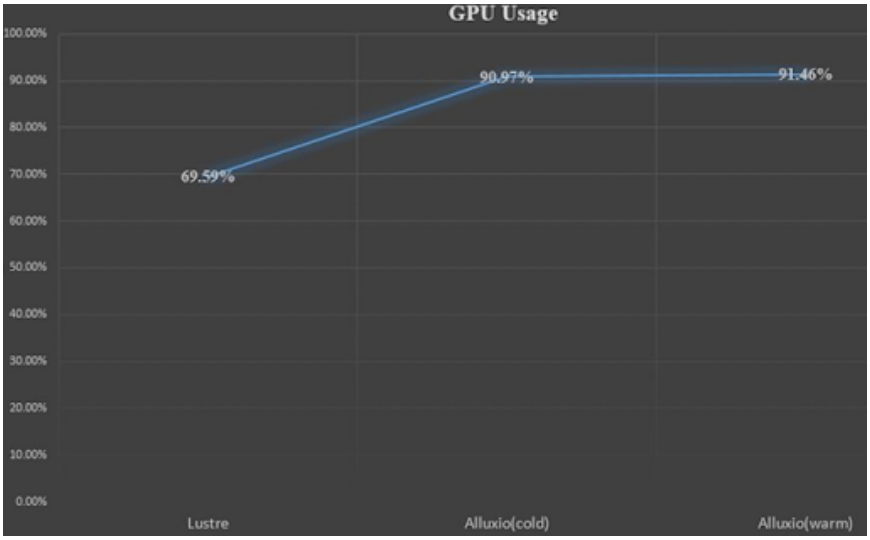
Optical Character Recognition (OCR)

In this test, we use a CRNN-based character recognition model based on the Pytorch framework. The data source is our self-collected 125GB image data converted into one large lmdb file. We made three benchmark tests, read directly from the underlying file system (Lustre), read from Alluxio without warmup (Alluxio, cold), and read from Alluxio with warm data (Alluxio, warm).



6 /

We found that the I/O traffic of the Alluxio warm cache node is reduced from 1300Mb/s to basically zero compared to the direct reading from the underlying distributed storage. This is a huge benefit to our platform. This is the most efficient and relatively cost-effective way to reduce the network traffic of the storage system without scaling the underlying storage hardware.



The average GPU usage of direct reading from the underlying storage computing is 69.59%, while for hot cache read, it increases to 91.46%, indicating that eliminating the I/O bottleneck can improve resource utilization for large file training jobs.

7 / Conclusion

By introducing the new architecture of Fluid + Alluxio, we have gained the following benefits:

- Speedup model training: through the test results, we can see a significant speedup on training jobs. By achieving data locality, the platform can avoid network bottleneck or resource competition, thereby effectively accelerating the data access in the model training process.
- Offload the underlying storage: by enabling local cache, the new architecture brings a better IOPS (Input/Output Operations Per Second), greatly reduces the workload pressure of the underlying storage system, and effectively improves its availability.
- Increase GPU cluster utilization: by efficient I/O read, the platform eliminates the data read bottleneck, and also avoids GPU idling while waiting for data, thus improving the GPU utilization rate of the entire cluster.
- Avoid I/O competition on the same node: the new architecture fully solves our previous pain points of I/O resource competition on the same node, the storage system I/O bottleneck, and the slow model training.
- More efficient cache management: the new architecture is a more cloud-native way to manage the cache. Engineers have changed from simply loading data in memory to managing and monitoring the cache. Kubernetes scheduling can perceive the cache and perform corresponding policy allocations, making jobs more efficient and easier to manage.

8 / Next Steps

Fluid + Alluxio has brought us a lot of benefits. We will keep working closely with the community and contribute to the following work in the future:

- We will continuously provide feedback to the community with more test results in more scenarios, and iterate and optimize the performance of Alluxio.
- We will test more data types, summarize findings, and provide best practices on performance tuning to the community.
- We will add an intelligent cache scheduling feature to Fluid.

About the Authors

Dongdong Lv, *Platform Architect*

- Many years of experience in cloud-native open source communities
- Responsible for ML platform architecture design, deep learning and AI modeling optimizations and accelerations
- Research areas: high-performance computing, distributed file system, distributed caching, etc

Qingsong Liu, *Algorithm Researcher*

- Responsible for the R&D of ML platforms and algorithms
- Research areas: cloud-native architecture, high-performance computing, voice and vision applications, machine learning algorithms, etc.