**Compressing and Archiving Logs to AWS S3 Glacier**

Abstract: Detail the problem

In distributed architectures, whether it be microservices or IoT device fleets consisting of hundreds of thousands or even millions of devices, there is often a need to compartmentalize the logs for each device in an S3 Bucket. The logs may be comprised of runtime application processed transactions, state or device metrics to be used for application intelligence such as billing, or more complex analytics used for performance and cost optimizations. In some applications, log files even serve as the ultimate source of truth and as such, there is a need for retaining logs permanently with high availability and durability guarantees while reducing the cost of storing these files in the Cloud. Many customers elect to either write device log files directly from their applications into AWS S3 in raw formats such as plain-text, JSON or XML or to stream logs to AWS facilities such as Kinesis Data Streams or Kinesis Data Firehose that can automatically create snapshots of the streams into log files and write them directly to an S3 Bucket. This blog post seeks to address the problem of archiving millions of log files to S3 Glacier for customers who choose to store logs permanently with the availability and durability guarantees provided by object storage services such as AWS S3 Glacier. That is to say, following a period of interest e.g. (one year), customers elect to move logs into cold storage to reduce operational costs associated with AWS S3 Standard.

Although AWS S3 provides automatic archiving features such as [Intelligent Tiering](https://aws.amazon.com/s3/storage-classes/) which automatically moves your files into less frequently accessible storage classes based on access patterns and S3 Bucket/Object [lifecycle policies](https://docs.aws.amazon.com/AmazonS3/latest/userguide/lifecycle-and-other-bucket-config.html), which can be configured to ultimately archive logs files into AWS Glacier depending on the customer’s use case. There may be some cost-related pitfalls that are worth considering before committing to the usage of these features, however. Rather than attempt to detail all the relevant use cases, we shall focus on one specific example that came directly from an AWS Customer and provide architectural guidance on how to design a logging construct with the aim of mitigating these cost-related pitfalls while adhering to the AWS Well Architected Framework

Use Case:

Consider a scenario where you have millions of application log files stored in AWS S3 Standard tier and decide that you would like these log files archived to AWS S3 Glacier cold storage in order to reduce operational costs. As previously noted, although S3 does provide bucket/object lifecycle policies that can automatically transition these log files for you into cold storage it is likely that this workload will end up costing a lot more than an optimized and strategically planned workload.

Example:

Suppose a customer has a business need to retain IoT device log files permanently but is unlikely to access these log files after a period of one year. Also, suppose an IoT device aggregates and writes 5 minutes worth of metrics to a log file as a single JSON entry. After 24 hours, the log file is written into a corresponding device sub folder within an S3 Standard bucket. This workflow produces 1 log file per day from each device in your fleet or 365 log files a year. It is easy to see that in a fleet consisting of thousands or even millions of devices, the number of log files written into S3 Standard over the course of one year starts to increase linearly by an order of magnitude as the number of devices increases by an order of magnitude. The number of files generated has an impact on the ops-related costs (PUT/POST/GET/LIST etc) on S3 Standard but also on the eventual transition of said log files into AWS Glacier. Consider the following graph which better highlights this specific example:

1 device 365 logs

10 devices 3650 logs

100 devices 36500 logs

1000 devices 365000 logs

10000 devices 3650000 logs

100000 devices 36500000 logs

1000000 devices 365000000 logs

Storage related costs for this use case also vary depending on the number of devices generating logs and the size of each log file. Although the number of devices and the size of each log file generated by a device ultimately drives this storage costs, there are two parameters worth considering that impact the actual log file size. The first is the size of each JSON entry in a 5 minute interval and the second is your application logging behavior. For example, devices may not necessarily log entries during periods of inactivity or down time, thus the size of each log file is likely to vary. Also, the size of each entry that is logged also impacts the overall file size and consequently the costs associated with storing these log files.

In order to fill in the details of this example, let us assume that each log entry has a size of 1KB and let us also assume that devices will always log an entry even though that may not necessarily be the case. Given that there are 288 five minute intervals in 24 hours, we can assume that the size of each log file will be 288KB on the high end. In terms of actual storage costs, this roughly translates into 288KB \* 365 = 105 MB per device, per year. Applying the cost model based on our previous assumptions, the storage consumed also increases by an order of magnitude as the number of devices you have in your IoT fleet increases. Consider the following graph which better highlights this relationship:

1 device 365 logs 105 MB ~ 100 MB

10 devices 3650 logs 1050MB ~ 1GB

100 devices 36500 logs 10500 MB ~ 10GB

1000 devices 365000 logs 105000 MB ~ 100 GB

10000 devices 3650000 logs 1050000 MB ~ 1 TB

100000 devices 36500000 logs 10500000 MB ~ 10 TB

1000000 devices 365000000 logs 105000000 MB ~ 100 TB

Although writing compressed log files will typically save a customer ~ an order of magnitude on their storage costs, it does not reduce the actual number of log files that are being written and will therefore eventually need to be archived. Also, archiving files smaller than 128KB is typically not very cost effective and as such, they are not candidates for auto archiving via object lifecycle policies. With that said, if log files generated per device do not meet this minimum threshold, the objects will need to be archived via Intelligent Tiering or they will need to be manually archived into Glacier by the customer. Both of these solutions have downsides: Intelligent Tiering has a higher cost margin than using lifecycle policies while manually moving objects to S3 is undesirable because it can negatively impact the operational excellence of the solution. Also of note is the fact that transitions between a storage classes such as S3 Standard to S3 Glacier will add ~ 32KB of size to each archive for metadata and tracking (archive description etc). For more details on S3 Glacier and it’s associated costs see the following FAQ Page: <https://aws.amazon.com/s3/faqs/>

In our example, we will be writing raw JSON log files in order to create a path toward using lifecycle policies and we will break down the costs associated with that approach. Consider that if we are writing compressed files (eg. gzip or bz2), the size of each log file in our hypothetical example will be ~ 28KB given that compression typically reduces file size by an order of magnitude. As noted above, files of this size are ineligible for automatic archiving via lifecycle policies. It would in fact cost you more to store the associated mandatory metadata for each file (~32KB) than it would to store your application log (~28KB). Thus, assuming no compression is used when writing these smaller files while also using lifecycle policies to archive them into S3 Glacier and subsequently also deleting the files from S3 Standard after archival, the cost using the AWS Cost Calculator can be broken down as follows: <https://calculator.aws/>

1 device 365 logs 105 MB ~ 100 MB Glacier Cost ~ negligible

10 devices 3650 logs 1050MB ~ 1GB $0.19

100 devices 36500 logs 10500 MB ~ 10GB ~ $1.86

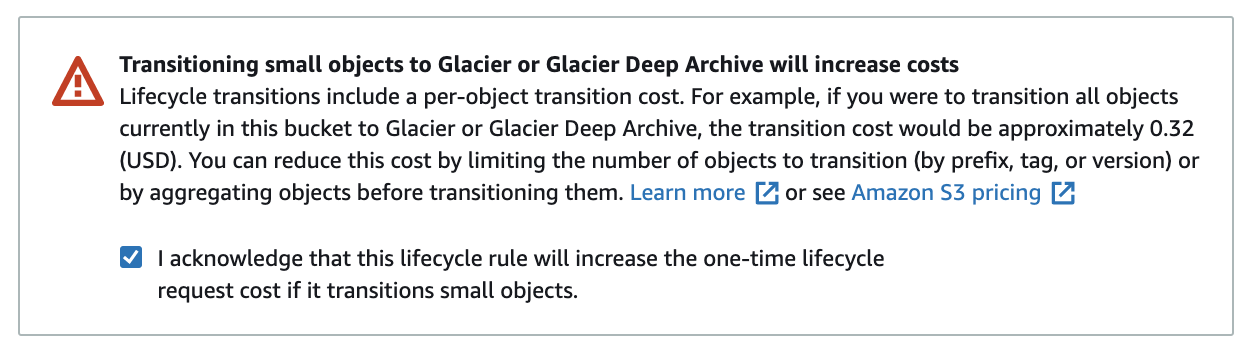
1000 devices 365000 logs 105000 MB ~ 100 GB ~ $18.65

10000 devices 3650000 logs 1050000 MB ~ 1 TB ~ $186.60

100000 devices 36500000 logs 10500000 MB ~ 10 TB ~ $1,860.60

1000000 devices 365000000 logs 105000000 MB ~ 100 TB ~ 18,659.60

For customers who have not considered archival cost implications with this type of use case, it is evident that creating a solution to this problem is not only highly desired but also warranted. Although storage costs in S3 Glacier are among the lowest in the industry, the task of getting a lot of smaller files into Glacier via lifecycle policies comes at a premium as highlighted above. The key question for customers then becomes how do I increase the size of each archive (and therefore reduce the total number of files) that I need to write into Glacier?



In addition to the cost calculator that can give customers a rough estimate of what it would cost to transition these types of files into Glacier, the lifecycle policy configuration also offers a potential solution to this problem in not only making customers aware of the cost implications associated with transitioning a large number of smaller files into Glacier but also providing a solution for how to reduce cost by way of data aggregation/compression. Also provided is an estimate of what it would cost to do so based on the individualized parameters of a customer’s workload, namely number of files to archive and size of each file to archive. In the example above, consider the cost savings realized by aggregating/compressing and archiving log files.

Suppose we will be generating archives for each device in the fleet containing logs over a period of one year and further compressing that archive in order to reduce our storage costs. Let us also assume we use tar as our archiving tool and gzip as our compression tool. The numbers here as it relates to cost start looking much better:

Starting:

1 device 365 logs 105 MB ~ 100 MB Glacier Cost ~ negligible - 1

10 devices 3650 logs 1050MB ~ 1GB $0.19 -

100 devices 36500 logs 10500 MB ~ 10GB ~ $1.86

1000 devices 365000 logs 105000 MB ~ 100 GB ~ $18.65

10000 devices 3650000 logs 1050000 MB ~ 1 TB ~ $186.60

100000 devices 36500000 logs 10500000 MB ~ 10 TB ~ $1,860.60

1000000 devices 365000000 logs 105000000 MB ~ 100 TB ~ 18,659.60

End Desired:

1 device 1 log archive (365 files) + gzip compression ~10 MB ~ free

10 devices 10 log archive (365 files each) + gzip compression ~105 MB ~ free

100 devices 100 log archives (365 files each) + gzip compressing 1050 MB ~ 1G ~ $0.01

1000 devices 1000 log archives (365 files each) + gzip compression 10500 MB ~ 10GB ~ $0.09

10000 devices 10000 log archives (365 files each) + gzip compression 105000 MB ~ 100GB ~ $0.90

100000 devices 100000 log archives (365 files each) + gzip compression 10500000 MB ~ 1 TB ~ $9.10

1000000 1000000 devices (365 files each) + gzip compression 10500000 MB ~ 10 TB ~ $90.60

As this data shows, although we are effectively reducing the overall cost of object transitions into AWS Glacier by more than 2 orders of magnitude, a significant consideration is the cost associated with actually generating the archives. Thus, a major consideration remains on how we can best optimize the archive generation process. E.G. (/listing/reading/aggregating/compressing/uploading) In the next section, we will explore a potential solution around “how to” reducing the number of archive objects that will ultimately end up in Glacier while at the same time ensuring that a customer retains all the log entries permanently.

Potential Solution:

A viable solution to this problem would be to read all of our log files written into S3 Standard, perform aggregation/compression processing on the records and then push them back to S3 Standard where they would now be eligible for transition into Glacier via a cost-effective lifecycle policy. Another option would be to write the archive files directly into Glacier via the AWS Console or CLI. This latter approach, however, would come with the added burden of customers having to maintain the Archive Name to Archive ID mapping and using it to query Glacier directly in order to retrieve archives of interest in the future. By utilizing lifecycle policies, we can effectively offload the management of Archive Name to Archive ID mappings to S3, enabling us to simply request an archive restore directly from the S3 service (Console/CLI) and allowing it to retrieve the archive from Glacier on your behalf. One caveat to this approach however is that there is a delay ranging from a few minutes to a few hours between the time the restore request is issued to the S3 service and when the archive will become available for download through S3 Console/CLI.

Best Practices:

1. Bucket Listing: In order to read log files back and build compressed archives we need to know which files to read back. This will require listing our target S3 bucket to obtain the files of interest. Listing S3 buckets has associated cost implications that can be mitigated by utilizing effective bucket partitioning, particularly for customers who utilize the same bucket for different workloads. That is, how you organize/compartmentalize/partition the files within your S3 bucket matters and it is a major architectural consideration when designing a cost-effective archival strategy. For example: Suppose that we are only interested in a particular class of log file residing within an S3 bucket that is used for various workflows. An effective archival partitioning strategy asks the question, if I want to list S3 for just this class of log file is there a way to avoid scanning the entire bucket? Using prefixes or partitions can dramatically reduce the costs associated with listing buckets in this case by avoiding the need to list all entries simply to find the interesting subset required.   Suppose an application is storing 5 minute metrics log files for each device, but it is also storing application error log files in the same bucket. A poorly partitioned bucket is defined as follows.   <Device ID>  <5m-device-metrics>  <metric files>  <application-error-logs>  <error log files>  Because the Device ID is unique this is a poorly partitioned bucket. If there were an interest in just archiving the device metrics, we would need to list all files just to find the ones of interest and dismiss the rest on the client side. Further, if we are simply interested in a particular year’s worth of metrics this strategy does not scale well because we effectively need to list everything in order to find the years of interest.   However, consider the optimized partition strategy defined here:  <Years>/  <5m-device-metrics>/  <device ids>/  <metric files>   <application-error-logs>/  <device ids>  <error log files>  With this optimal partition strategy, we can selectively specify a listing prefix to list a particular year’s worth of metrics from a particular device or from all devices without having to also scan the application error log files as they are not included in the prefix of interest. This reduces the number of listing operations into the bucket and thus reduces the overall cost associated with listing.
2. Workload Contexts: Have an effective processing architecture that will optimize bandwidth, performance and runtime. Consider organizing workloads into contexts where a context consists of one device containing all the log files to be archived over a desired time interval or size interval.
3. Idempotent Workloads: In a producer/consumer architecture where the producer is generating the work contexts and the consumers are processing said work contexts, it is imperative that you have an idempotent workload, particularly in the case where we have multiple consumers within the same compute node (multi-threaded) or multiple consumers spread over multiple compute nodes (distributed system). That is, processing the same workload context multiple times should have no impact on the end state (desired result). Depending on what technologies, archive naming conventions or bucket partitioning you choose to use, this matters and deserves sufficient consideration.
4. Common Message Bus: In a pull producer/consumer architecture consider using a queuing technology such as Amazon SQS. In a push producer/consumer architecture consider using a notification service such as Amazon SNS. These technologies will impact idempotent considerations and should be considered carefully. When used effectively these technologies provide added value on architecture robustness particularly around Resiliency and Error Path.
5. Writing Archives: Consider using Multi-Part uploads for archives that are large in size as this will reduce the overall time it takes to get archives committed back into S3 after they are built. For example, writing a 10MB archive takes longer than writing 5 2MB parts concurrently at a tradeoff to network bandwidth.
6. Cleaning Up: S3 objects that have been committed to an archive and consequently uploaded back to S3 should be deleted from the source bucket in order to reduce S3 Standard costs. This should be done if and only if the archive was generated and committed successfully back to S3. Certain programmatic measures should be taken to ensure that this is the case. Objects should never be deleted from S3 before the Archive has been generated and uploaded successfully! This is especially important in asynchronous applications. In addition, the integrity of the file may be verified using the object ETAG which is effectively and MD5 Sum of the object.
7. Error Path: Ensure that your architecture is fault tolerant by preserving state so you will not have to restart the process in the event of a failure or a crash as this will increase overall costs and the time it takes to complete your workload. Rather, in the event of a failure, an efficient architecture picks up where it left off and resumes the workload through to completion.
8. Compute Instances: Choose optimal instances for your workloads in order to reduce cost. For processing of smaller files (reading objects/writing archives) choose Compute instances that are processing/memory optimized. Network bandwidth should not be a major factor on reading smaller files from S3 as the compute node will likely be processing/memory bound. Network bandwidth may become an issue as larger archives are being uploaded back to S3 particularly via use of multi-part uploads. It is recommended that the AWS S3 Client is launched with a high number of threads in order to prefer high concurrency on downloads over bandwidth. With this approach however, bandwidth constraints may become an issue as you upload generated archives back to S3.
9. Processing: Run your solution on AWS Compute to reduce both processing time and bandwidth charges associated with reading from S3 to the public internet! If using an optimized compute instance tailored toward your workload it should improve both processing efficiency and overall cost.
10. Test: Be sure to test your solution thoroughly via a representative workload before using it. Construct an effective testbed that can be used for benchmarks. Take note of the resource consumption on CPU/Memory/Disk/Networking and use this information to select optimized compute instances that will end up reducing cost.

Tips and Additional Considerations:

1. Reusability: Don’t invest in a robust solution unless you can reuse it.
2. Consultation: Reach out to a certified AWS Partner or your solutions architect to assist with solution design.

Future Considerations:

Asking the right questions is critical to optimizing your workloads to ensure that a clear and cost-effective path toward archiving exists. For example: can the application logging behavior be modified in order to yield larger files thus reducing the costs associated with archiving a high number of smaller files? If the answer isn’t clear or requires a high cost to implement effectively then the second order questions to consider are how can you partition your workloads better to reduce costs associated with aggregating/compressing logs. It is good practice to invest in an efficient tool that you are likely to use again and make sure it is designed the right way. Talk to your Solutions Architect, an AWS Certified Partner, or AWS Professional Services who may help with design considerations. In addition, ask if there are ways to optimize your workloads further to ensure a clear and cost-effective path toward archiving exists.

Architecture and Source Code:

<https://github.com/iiljazi/s3ArchiveBuilder>