# Cloud-based Big Data Serverless Reference Architecture for Data Mining & Analytics

Kaustuv Kunal [0000-0001-6110-1321]

littilabs.com, India
kaustuv.kunal@gmail.com

**Abstract.** Serverless architectures are cost effective, fast to market, reliable and less maintainable. With the evolution of public clouds, their usage has gained momentum in software industry specially, for big data processing. However, building such systems is challenging task particularly, for start-ups. The paper proposes a baseline serverless, scalable, end-to-end batch log processing architecture for data analytics and modeling followed by its case study. The four-layer Function-as-a-Service reference architecture is effective, distributed, self-maintainable, inexpensive and shall be setup leveraging any public cloud. Additionally, it aids in data management, security and stakeholder's profiling.

**Keywords:** Big Data Processing, Cloud Computing, Serverless Architecture, Batch Processing, Log Processing, Public Cloud, Reference Architecture.

## 1 Introduction

Big data have received substantial attention over the years, specially due to Internet and exponential growth of digital data. Every sector from agriculture, commerce and healthcare to governance uses big data. Nowadays, petabytes of data traffic is common for several mobile applications. Big data needs special architecture and cannot be processed using traditional infrastructure. Generally, such architectures are distributed and include collection, processing, analysis and visualisation. Initially, Google File System (GFS) [5] based distributed architectures, for instance, Hadoop [6] and its ecosystem [7,8,9,10] gained lots of popularity, a reason for this was their usage of commodity hardware. Dedicated in-house setup is not feasible for many enterprises. With evolution of public clouds, such as [2,3,4], infrastructure setup became faster, easier, flexible, cost effective and scalable. Public clouds provide multiple resources and components for building full-fledged processing architecture.

In public cloud, generally compute instance is costlier than storage instance. Serverless architectures [11,50] are pay-as-you-go models as they do not reserve any computation instance. Obviously, there is no upfront provisioning or management of servers require. Besides reducing infrastructure cost and reducing operational cost, such architectures are also scalable and fast-to-market. Two major categories of serverless architectures are Function-as-a-Service (FaaS) [1,50] and Backend-as-a-Service (BaaS). BaaS are mostly third-party backend services such as, user authentication,

hosting etc., whereas, FaaS architecture runs on stateless compute containers and explicitly uses functions as the deployment unit.

Categorised on latency, applications are either batch or real-time. Both have different architectural requirements and own significance. Unlike real-time applications, batch applications do not have stricter latency requirements. They are useful in reporting, analytics, modeling etc., for example, log analytics. Designing a serverless batch architecture is a complex task. Organizations, specially those with no prior experience, spend significant amount of time and resources on this.

The paper proposes a baseline FaaS batch processing architecture. This sequential layered architecture can be deployed leveraging any public cloud provider. Proposed baseline is useful for any mid or small size companies and institutions, trying to set up cost-effective, scalable, modular big data batch processing system on public cloud. Further, a use case implementation of ecommerce campaign log analytics, deployed on AWS, using proposed architecture is discussed.

The structure of the paper is as follows. Related work is discussed in section two followed up with problem background in section three. The proposed baseline architecture is discussed in detail in section four, followed by its implemented case study in section five and performance evaluation in section six. Finally, paper concludes in section seven

## 2 Related Work

Lambda architecture [35] combines batch and streaming pipelines into single architecture. It primarily consists of three layers. First, batch layer for distributed processing. Second, serving layer for incrementally indexing the batch layer and third, speed layer, for complementing the serving layer by indexing the latest data. Kappa Architecture [36] is mainly a streaming system, extendible for historical batch processing as well. Pipeline [37], Event-Driven [38] and Microservices [39] are few other architectures types frequently leveraged by big data processing applications.

Several big data reference architectures have been proposed in past. Based on use case architectures of Facebook, Amazon, Twitter, and Netflix a four-layer abstract reference architecture was proposed [41] and further mapped to LinkedIn [42]. A technology independent reference architecture for big data systems was proposed based on published big data use cases and associated commercial products [45]. Domain specific processing architectures and frameworks for, public transportation [43], cluster Monitoring [44], railway asset management [46], manufacturing [47] and Artificial Intelligence [48,49] have also been suggested.

Big Data serverless systems on specific public cloud and entities have been designed. An architecture [52] on AWS lambda and Apache Spark libraries and another one [55] on IBM cloud functions have been presented. In [53] an event-driven serverless architecture based on OpenWhisk [23] and Kubernetes [54] is suggested. In [52] multi-cloud distributed system is advocated in order to overcome cloud specific serverless entity limitation. Literature review shows that there are limited reference architectures in the big data context as well as lack of concrete or coherent reference commercial

serverless architecture. This demonstrates a need for further research in serverless reference architecture for big data systems.

## 3 Background



It is widely adopted practice to record runtime information into logs. Increase in application usage and number of hits, leads to the growth of the volume of logs. To handle large volumes, efficiently and effectively, intelligent log analytics platform is needed [34]. Due to their high volume and high verocity, logs are candidate for big data processing. Further, logs are mostly time stamped hence useful in debugging and analysis. However, log analysis poses many challenges. Few among them are mention below.



Storage: Logs are generated continuously with exponential growth in size. Owing to which, log servers are susceptible to failure for instance, out of memory error. Therefore, storage and fulfilment of memory requirements is one of the concerns.

Redundancy: Significant part of the log data are not beneficial for business. Also, on multiple occasions entries can be redundant. Clearly, segregation of business relevant data is a challenge.

Structure: Log structure changes with time. This can be because of internal factors for instance, application code modifications or external factors such as, protocol updation. Smooth adoption of the log structure modifications is definitely a challenge.



Processing: Many logs are readable text data. Due to their non-serialized nature, raw logs are not much suitable for fast processing. Furthermore, multiple processing requirements exist for the same data and multiple jobs require to access same data.

Sources: Log data can arrive from multiple sources such as sensor, mobile app, websites etc. Ensuring timely collection of all logs from heterogenous sources is a challenge.

Stakeholders: Multiple stakeholders access the same system. Hence, intelligent access management and user profiling is also a challenge for big analytics systems.



Cloud computing is capable to manage the high production rate, large size and diversity of log files. Also, it is a field proven solution that is used for years by many big companies like Facebook, Google, Microsoft, Amazon, eBay, etc., to process logs. By the aforementioned, we can assume that log analysis is a Cloud use case. One approach of cloud computing that has gained traction in both industry as well as research community is serverless big data processing based on FaaS platforms. FaaS platforms enable developers to define their application only through a set of service functions, relieving them of infrastructure management tasks, which are executed automatically by the platform.



## 4 The Architecture

This baseline end-to-end cloud-based architecture is specifically designed to process time stamped log data across variety of domains. FaaS based, four layers modular architecture entities options are presented. Input to our layer architecture is log server

(or any storage dataset) and output shall be analytics or modeling backends. Data are usually stored in time-stamped folders. Every layer is responsible for a specified function. Layers are discussed in first part of current section. The architecture is built utilizing three major cloud components first, storage units second, compute instances and third, messaging channels, these are discussed in the second part of current section.

### 4.1 The Layers

Four sequential layers namely, fetch, transform, process and customize are depicted in Fig.1. Each layer is targeted to act as a server, performs a dedicated functional task and passes the processed data to the subsequent layer. Layers shall be thought of as independent extract-transform-load (ETL) units. Additionally, each layer will process its time unit data concurrently.


**Fig. 1.** Four sequential FaaS layers

**Fetch Layer**

In the first layer, data will be fetched from external log server without any modification. For this, periodic polling will be followed by simple copy. Last n folders of timestamped log data (stored as time-labelled folders), will be polled concurrently for any new arrival. Every log folder represents a unit time, for example hour. Value of n should be large enough to cover any delays and out-of-memory errors. An obvious approach to decide n is, assign it to the maximum delayed arrival. This is depicted in equation-(1).

$$n = \max(l) \tag{1}$$

Here, l is the measured list of data latencies. However, n may take large enough value. A better approach would be by utilising box plot boundary values. A quartile-based formula, for measuring n, is given in equation-(2). Here, Q3 and IQR are third quartile and interquartile range respectively.

$$n = Q3(l) + IQR(l) * 1.5 \tag{2}$$

Primarily, Fetch layer provides the architecture, independence from external systems for instance, log servers. It ensures that in case of any data mismatch, debug and validation requirements, system shall have enough data. Further, I suggest a dynamic mechanism by recording data arrival details in a meta file and a scheduled job to

periodically compute quartile and n value. Fetched data shall be stored in bucket-based cloud storage system. Various storage choices are discussed in the first part of the next section under storage unit. Additionally, tasks such as conversion of heterogenous logs into homogeneous one, elimination of duplicate entries, pre-ingestion check and post-ingestion checks shall also be provisioned in this section.



**Transformation Layer**

After Fetch, Transformation layer is invoked. Objective of this layer is to transform raw logs for faster processing. Plain row logs are slower to process, converting them into machine readable binary format will enable faster processing. Conversion into binary format is known as Serialization. Indexing and compression are few other transformation techniques. Multiple data formats are available for transformation. Format selection is based upon several factors such as, properties of log, processing objective etc. Several conversion formats exist, lets briefly look into few.



Parquet: Introduced by Google engineers in the Dremel paper [13], Parquet [14] is column storage format that stores nested data structures in a compressed, flat columnar format. Data stored in a Parquet file is described by a schema, which resides at the footer. Parquet files are compressed, splitable and fast to process. Also, they are optimized for the Write Once Read Many (WORM) paradigm that makes them slow to write, but incredibly fast to read. It is good choice for read-heavy workloads specially when accessing only a subset of the total columns.



Avro: Originally a data serialization system. Avro [15] is a row-based and splitable data format. Its schema is stored in json while the data is in binary. Contrary to parquet, Avro stores schema at the header. It provides robust support for schema evolution by managing added fields, missing fields, and updated fields. Also, with Avro no coding generator is needed by data exchange services to interpret data definition and to access data. Data stored with Avro can be shared by programs using different languages.



ORC: Originally created to store hive tables for efficient write and faster read, ORC [16] is a columnar format like Parquet. ORC, abbreviated for Optimized Row Columnar, stores data compact and enables skipping over irrelevant parts without the need for large, complex, or manually maintained indices. Also, it supports ACID properties.

In order to sort format selection dilemma, factors such as log structure, processing type, data evolution shall be considered. For nested structures, parquet, Avro or json are suitable. Parquet is preferred if primary requirement is compression whereas, Avro has best support for schema evolution. Also, for selective processing such a SQL queries, columnar format is better, whereas for ETL processes, row format is preferred. Additionally, compressed techniques (snappy, LZ4, etc.,) can also be applied in this phase though these shall be strictly on case-to-case basis. Importantly, this layer is optional layer and can be ignored for instance, if raw logs are already pre-formatted.



**Processing Layer**

Once transformed, next step is to process data in order to fetch relevant information. Data processing layer is the heart of this architecture and responsible for core big data

processing tasks. Various big data processing frameworks are available. However, metastore based frameworks, for example hive, are not suitable for serverless compute instances. Data shall be processed inside compute units leveraging processing frameworks. Compute instance shall be occupied only till the execution. Let's briefly look into few of the processing frameworks.



Spark: Written in Scala, Spark [17] is ideal processing framework for iterative jobs. It treats distributed data as a resilient distributed dataset (RDD) object, linked through DAG and has ability to recover lost partition using lineage technique. It also supports a rich set of higher-level tools including Spark SQL for structured data processing, ML-lib for machine learning, GraphX for graph processing, and Structured Streaming for stream processing.

MapReduce: One of the initial processing frame work of Hadoop. MapReduce[56] is key-value based, two-stage processing framework and a kind of divide and conquer technique.



Cascading: Cascading [18] is a java-processing library. Like MapReduce, Cascading is also a divide and conquer approach but with a different metaphor. It thinks of user data as stream and uses taps, pipes and sinks abstraction. Data is stream of tuple. Tuple is ordered list of fields. Tuples travel inside pipes from sink. Pipes allow parallel execution. Combination of pipes creates a flow. Cascading job is DAG of flow.

Beam: Apache Beam [19] is an open source, unified model for defining both batch and streaming data-parallel processing pipelines. It is particularly useful for parallel data processing tasks, in which the problem can be decomposed into many smaller bundles of data that can be processed independently and in parallel such as pure data integration and ETL. Such tasks are useful for, moving data between different storage media and data sources, transforming data into a more desirable format and loading data onto a new system.



NiFi: Originally, built to automate the flow of data between systems, Apache NiFi [20] supports powerful and scalable directed graphs of data routing, transformation and system mediation logic.



The above list is not exhaustive and plethora of other processing frame workings exists for instance, Flume, Tez, etc. The best place to know them is their respective project websites. In addition to log structure and processing requirements such as, mining, retrieval, prediction etc., an important factor to consider while deciding on processing framework is the availability of programming resources in the organisation.

**Customization Layer**



Customization Layer interacts with external systems. In this layer, processed data is further customized for micro requirements of report generation, modeling, sampling, visualization etc. Commonly, same data are required by multiple frontends, for instance database, visualization, analytics, data science modeling etc., with a slight variation. This layer fulfils such requirements by generating data varieties and granting stakeholder's access. Further, the layer transfers data either directly to frontend or to the respective backend. Moreover, for faster setup and tuning, client specific metadata shall be attached. The logics can be written in any programming or scripting language.

### 4.2 The Components

Proposed serverless architecture is depicted in Fig. 2. It has 3 main components namely, storage unit (SU), computation unit (CU) and Messaging queue (MQ). We shall look into these one at a time.



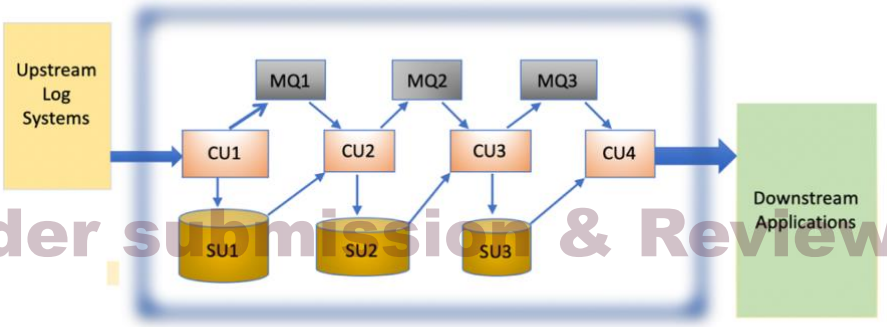




**Fig. 2.** Architecture Overview



**Storage unit (SU)**

Data is stored in storage units. Each layer, except layer-4 i.e., customization, has a dedicated storage unit. Some desired characteristics of storage units are rich APIs support, archival facility, self-hosting ability and cost-effectiveness. Bucket-based storages are preferred in serverless systems. Support for regular data compression is important because once transformed, processed or customized, data shall be required only during validation and debugging. Next, we shall look into some available options for storage unit in public cloud space.



AWS-S3: Pioneer in providing object storage, Amazon Simple Storage Service (S3) lets user upload any amount of data and takes the burden of storage limitation away. Inside S3 we have provisions for storing low latency data (S3-standard) and archive storage (S3- Glacier). Lifecycle management data can be easily transferred from one storage (S3) type to another. Also, it has rich API support. However, user must be aware about its security-based practice and regions.

GCP - Cloud Storage Buckets: A storage and retrieval system by Google cloud, it is supported by Google's own reliable and fast networking infrastructure to perform data operations in a secure and cost-effective manner. It has 4 performance tiers, standard, nearline, codeline and archive.



Azure- Blob storage: Blob Storage allows storage and retrieval of massive amount of unstructured, scalable data as Binary Large Objects (BLOBs). It also facilitates archival storage tier. Hot storage, for the frequent accessed data and cool storage tier and archival storage tier, for rare accessible data.

IBM's storage [21] and Oracle storage services [22] are few other notable bucket storage options.

**Computation unit (CU)**

Computation units are core of proposed serverless architecture. They are unreserved compute instances that execute code after fetching required data from object buckets. Each layer of architecture has intended compute unit. Dynamic memory allocation, auto scaling & descaling and support for microservice architecture are few desired qualities of computation unit. These are pay-as-you-go instances and should ideally charge only for the compute time. We shall see few compute component candidates.

AWS Elastic Beanstalk: A PaaS offering from AWS, Elastic Beanstalk takes care of provisioning, load balancing and scaling. It has strong support for docker containers and allows easy deployment of applications. Moreover, it supports many server environments including Apache HTTP Server, Nginx, Microsoft IIS, and Apache Tomcat.

AWS Lambda: AWS Lambda lets us run code without provisioning or managing servers. With Lambda, we can run code for virtually any type of application or backend service at scale with zero administration, without exposing code at Rest end points and pay only for the compute time consumed. AWS Lambda is one of the first microservices environment on the public cloud. Lambda is simpler and less expensive than elastic beanstalk.

GAE (Google App Engine): GAE is Google-cloud's fully managed serverless application platform with capabilities such as automatic scaling-up & scaling-down of applications, fully managed patching and management of servers, load-balancers and built-in mem-cache. App Engine is designed to manages all infrastructure concerns.

GCP Cloud Functions: A scalable, pay-as-you-go FaaS platform to run code with zero server management and with no server provision, manage, or upgrade. It allows automatic scaling based on the load along with integrated monitoring, logging, and debugging capability, built-in security at role and per function level based on the principle of least privilege and bills only for function's execution time metered to the nearest 100 milliseconds. Also, it can run functions across multiple environments and prevent lock-in.

GCP Cloud Run: A serverless compute platform that enables to run stateless containers, invocable via HTTP requests. Cloud Run is available as a fully managed and pay-only-for- what-you-use platform.

Microsoft Azure Functions: It is a serverless computing service hosted on the Microsoft Azure public cloud. The Microsoft Azure service allows us to run small pieces of code in Node.js, C#, Python, PHP and Java and enables running application code on-demand without requiring the provision and management of the underlying infrastructure.

Microsoft Azure - App Service: Quickly build, deploy and scale web apps and APIs on user terms. Also, an opensource compute entity worth exploring is Apache's OpenWhisk.

**Message Queue (MQ)**

Message Queues are communication units and primarily used for triggering the compute unit. It also keeps track of execution state and status of time-stamped folders. Every time-stamped folder has a specified MQ. They are processed concurrently. In,

this asynchronous communication, each message is processed only once by a single consumer i.e., compute instance therefore, messaging pattern is often called one-to-one or point-to-point communications. MQ aids in scalability, testability, maintainability, flexibility of the overall architecture. We shall look into few of the candidates for MQ. This list obviously is not exhaustive.



Kafka: Apache Kafka [24] is an open sources event streaming platform. It is used for messaging, stream processing, data integration, and data persistence. It is designed for very high throughput and can process thousands of messages per second. However, it is more suitable for stream processing.

Rabbit-MQ: RabbitMQ [25] is lightweight and easy to deploy on premises and in the cloud. It supports multiple messaging protocols. RabbitMQ can be deployed in distributed and federated configurations to meet high-scale, high-availability requirements. It runs on many operating systems and cloud environments, and provides a wide range of developer tools for most popular languages.



Amazon SQS: Amazon Simple Queue Service (SQS) [26] is a fully managed message queuing service by AWS that enables developers to decouple and scale microservices, distributed systems and serverless applications. It eliminates the complexity and overhead associated with managing and operating message-oriented middleware and empowers developers to focus on differentiating work. SQS is not as fast as Kafka also, it does not fit to high workload. It is more suitable for eventing, where count of events per second is not so much. SQS offers two types of message queues Standard and FIFO. Standard queues offer maximum throughput, best-effort ordering and at-least-once delivery. FIFO queues are designed to guarantee that messages are processed exactly once, in the exact order that they are sent.



Google Cloud Pub/Sub: A scalable, durable, event ingestion and message delivery system that allows creation of infrastructure to handle message queues. Pub/Sub [27] delivers low-latency, durable messaging by using two core components that is, topics and subscriptions. You can create a topic where messages will be sent to and subscriptions attached to the topics. All messages sent to a specific topic will be delivered to all the subscriptions attached.



GCP Cloud Tasks: Cloud Tasks [28] is a fully managed queuing system service for Google Cloud Platform for managing the execution, dispatch, and delivery of a large number of distributed tasks. It can perform work asynchronously outside of a user or service-to-service request.

Azure Queue-Storage: An entity in the Azure cloud infrastructure. Queue Storage [29] allows storage of large numbers of messages, that can be accessed from anywhere in the world via authenticated calls using HTTP or HTTPS. A queue message can be up to 64 KB in size. A queue may contain millions of messages, up to the total capacity limit of a storage account. Queues are commonly used to create a backlog of work to process asynchronously.



Azure Service Bus: Queues are part of a broader Azure messaging infrastructure [30] that supports queuing, publish/subscribe and more advanced integration patterns. They are designed to integrate applications or application components that may span multiple communication protocols, data contracts, trust domains, or network environments.

Other components of architecture are local or remote job invocation machine (such as EC2), remote procedure call (RPC) and continuous delivery & continuous deployment (CI/CD) mechanism such as Jenkins [33].

## 5 Case-Study

Presenting a case study that is implemented leveraging proposed architecture. This can aid as a reference architecture as well as in understanding of overall architecture with related functionality and data flows.

Advertisers need their campaign's insights to tune the campaign further. Data generated from campaigns are large enough and suitable to process using big data technologies specially for analytics and modeling. When user clicks on the ad, her activities are logged into the campaign log server. Publisher's Ad can be evaluated on various parameters such as location, device, OS, user profile etc. Further, prediction shall be required for various metrics such as CTR (click-through-ratio), CPM (cost-per-mile), CPC (cost-per-click) etc. Using proposed architecture, a FaaS based campaign analytics system was developed and deployed on AWS. Each campaign data is processed parallelly and independent of each other. AWS -S3 and AWS-lambda are storage unit and compute unit respectively. Our queue data requirement was not much hence we found AWS- SQS suitable as well as cost effective. It contains meta data that majorly includes, time (year, month, day, hour), layer, publisher and campaign information. Overall implementation is depicted in Fig. 3. Next, we shall look into the 4 layers.

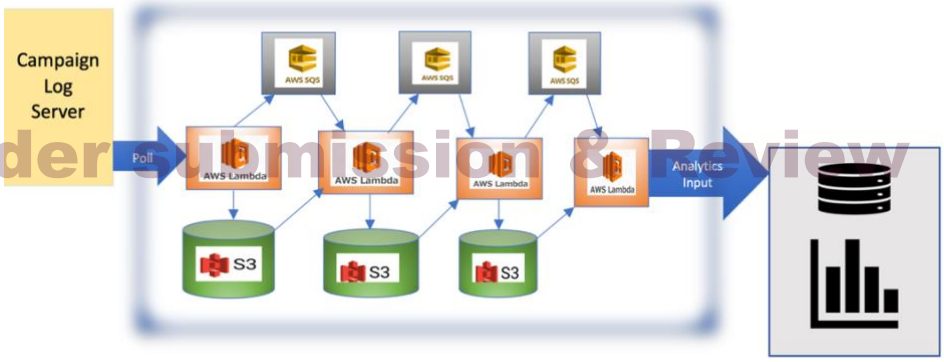


**Fig. 3.** Architecture Implementation Using AWS

### 5.1 Fetch

Live data is dumped into log server in hourly data-folders. Log servers are EC2 instances. They are polled for last 7 day's new arrival via Spring based rest framework. The polling job is scheduled to run every hour. Any new arrival is first, dumped into AWS bucket and second, an SQS message is generated. Jobs run inside AWS lambda.

It takes care of provisioning, scaling and runtime management. SQS message contains metadata and mainly specify time-stamped campaign data folder and intended invocation layer.



### 5.2 Transform

SQS messages generated in fetch layer invoke transform layer. Jobs runs in serverless compute instance AWS lambda, and generate the S3 path from message content. Further, jobs fetch the plain log data from the path and initiate transformation. For data formatting, Avro is our preferred choice. This is mainly because of two reasons. First, due to Avro's robust support for schema evolution and second because of Avro's compatibility with json (as majority of our logs structure are JSON). Moreover, no field preference is specified by client therefore, all the log field are treated equally important so columnar storage is not preferred. Once transformed, Avro files are dumped to separate transformation S3 bucket. Finally, SQS message is generated for transformed time-stamped folder and further invokes next subsequent layer.



### 5.3 Process

Flow of this layer is similar to transform. However, input bucket would be transformed S3 bucket and output would be processed bucket. Depending on processing requirement, we have mainly used 3 type processing frame work, MapReduce, Cascading and Spark. Spark is fastest to process, however due to high memory requirement it is costliest. MapReduce is one that initially met our requirement with reasonable cost. Cascading has reduced development cycle due to its lesser line of codes [31]. In this layer, microservice architectures, such as CloudBreak [32], are also found to be very useful for job execution and transferability. Majority of the produced output data are .csv file. Primarily jobs are summarization jobs, that summarize data across various sectors, for instance, location, model, publisher, operator, referrer, etc., based on time factors such as, daily monthly, yearly etc.



### 5.4 Customize

Once transformed data is processed and stored in respective bucket, customization layer is invoked. Multiple clients facing jobs execute in our serverless compute instance. Primarily, they provide input to two categories of applications. First, analytics system for campaign data visualisation and second Data Science (DS) applications for predictive analysis. For analytics dashboard, summarized data is dumped to MySQL. Jobs are written in java using JDBC. DS applications consume csv file from S3 for predictive analysis. DS jobs are written in Python. Further, this layer can be accessed by many developer groups and stakeholders. Multiple data level access control groups are defined here. This aids in efficient data access, management and security. Furthermore, for code management and version control Git is used. Maven and Jenkin are opted for build and CI/CD respectively.

## 6 Performance & Limitations

Our deployed system successfully supported more than two dozen live ad campaigns. The largest campaign generated close to a million records per day. Initially for long running applications, serverless compute instances of process layer threw timeout error and at times faced hanging issue. Though issue was resolved, after modifying the code and enabling restartability feature. Furthermore, checkpoint mechanism was injected to tackle any data loss issue. Due to client's non-disclosure agreements AWS component configurations, finer data processing specifications and performance evolution details has not been discussed.

In order to capture any late data, data is being polled for last n days, where the default n being seven i.e., a week. For any delayed files whole folder need to be reprocessed, this sometimes leads to redundant and multiple processing. This shall be eliminated, by introducing transaction-ids mapped with each time-stamped folder as a preprocessing step and thereafter, in case of any duplicate transaction-id, by dropping the older one. However, not much consequences were observed and reported due to this limitation. Mainly because, hourly folder size is not much (2-80 MB) and processing time for folders generally do not go beyond few minutes. Hence, this feature was made optional. Furthermore, as the architecture is intended for public cloud, it is advised not to use it for classified data applications such as, banking. Additionally, obviously system is prone to typical serverless architecture drawbacks [40] such as, no mechanism for optimizing data transfer between compute and storage instances.





## 7 Conclusion

The paper proposes a big data log processing cloud based serverless architecture for On line Analytical Processing (OLAP). Various architecture components and layered techniques are discussed and compared. Further, a case study implementation leveraging proposed architecture in ecommerce campaign analytics on AWS is discussed. It is clear that, the FaaS based, 4-layer modular architecture can be applied in broad range of ETL tasks and produce coherent results. Organizations looking for a cost effective and quick infrastructure or proof-of-concept setup on public cloud shall benefit from this baseline architecture. It has been shown that in addition to typical serverless advantages our multilayer pipeline architecture is self-maintainable, scalable and dockerized. It aids in access control, profile management and data security. Future work includes developing baseline integrated architectures for real time processing and also for data science and deep learning frameworks.



## References

1. M Eriksen, Your Server as a Function, Proceedings of the 7th Workshop on Programming Languages and Operating Systems, 5:1--5:7, (2013).
2. Amazon-Web-Services, https://aws.amazon.com/, last accessed 2021/08/05.

3. Google Cloud, https://cloud.google.com/, last accessed 2021/08/05.
4. Microsoft Azure, https://azure.microsoft.com/, last accessed 2021/08/05.
5. Ghemawat Sanjay, et al., The Google File System. Proceedings of the nineteenth ACM symposium on Operating systems principles, Bolton Landing, NY, USA, (2003).
6. Apache Hadoop Project, http://hadoop.apache.org/, last accessed 2021/08/05.
7. Apache Hive Project, https://hive.apache.org/, last accessed 2021/08/05.
8. Apache HBase Project, https://hbase.apache.org/, last accessed 2021/08/05.
9. Apache Oozie Project, https://oozie.apache.org/, last accessed 2021/08/05.
10. Apache Sqoop Project, https://sqoop.apache.org/, last accessed 2021/08/05.
11. Baldini, et al., Serverless computing:Current trends and open problems, arXiv:1706.03178v1, (2017).
12. Kuhlenkamp, et al., An evaluation of faas platforms as a foundation for serverless big data processing. In Conference on Utility and Cloud Computing, UCC'19, pages 1--9, New York, NY, USA, (2019).
13. S. Melnik et al., Dremel: interactive analysis of web-scale datasets. Proceedings of the VLDB Endowment, Volume 3, Issue 1-2, (2010).
14. Parquet Project, https://parquet.apache.org, last accessed 2021/08/05.
15. Apache Avro Project, https://avro.apache.org, last accessed 2021/08/05.
16. ORC project, https://orc.apache.org/, last accessed 2021/08/05.
17. Spark Project, https://spark.apache.org, last accessed 2021/08/05.
18. Cascading, https://www.cascading.org, last accessed 2021/08/05.
19. Apache Beam, https://beam.apache.org/, last accessed 2021/08/05.
20. Apache Nifi Project, https://nifi.apache.org/, last accessed 2021/08/05.
21. IBM Object Storage, https://cloud.ibm.com/docs/containers?topic=containers-object_storage, last accessed 2021/08/05.
22. Oracle Object Storage, https://docs.oracle.com/en-us/iaas/Content/Object/Concepts/objectstorageoverview.htm, last accessed 2021/08/05.
23. Apache OpenWhisk Project, https://openwhisk.apache.org/, last accessed 2021/08/05.
24. Kafka Project, https://kafka.apache.org/, last accessed 2021/08/05.
25. RabbitMQ, https://www.rabbitmq.com/, last accessed 2021/08/05.
26. Amazon-SQS, https://aws.amazon.com/sqs/, last accessed 2021/08/05.
27. GCP- Pub/Sub, https://cloud.google.com/pubsub/, last accessed 2021/08/05.
28. GSP- Tasks, https://cloud.google.com/tasks, last accessed 2021/08/05.
29. Azure - Storage Queues, https://azure.microsoft.com/en-in/services/storage/queues/, last accessed 2021/08/05.
30. Azure - Service Bus Messaging, https://docs.microsoft.com/en-us/azure/service-bus-messaging/, last accessed 2021/08/05.
31. K Kunal, Analysing Cascading over MapReduce**,** proceedings of International Journal of Computer Applications (IJCA), 9(1): 1–5, November 2016).
32. CloudBreak Project, https://docs.cloudera.com/HDPDocuments/Cloudbreak, last accessed 2021/08/05.
33. Jenkin, https://www.jenkins.io/, last accessed 2021/08/05.
34. Z Zheng, et al., System log pre-processing to improve failure prediction, IEEE/IFIP International Conference on Dependable Systems & Networks, (2009).
35. Nathan Marz, Big Data Principles and Best Practices of Scalable Realtime Data Systems, (2015).
36. Kappa Architecture, http://milinda.pathirage.org/kappa-architecture.com/, last accessed 2021/08/05.

37. CV Ramamoorthy, HF Li, Pipeline Architecture, ACM Computing Surveys (CSUR), (1977).
38. Brenda M. Michelson, Event-Driven Architecture Overview, Patricia Seybold Group, (2006).
39. A Balalaie, et al., Microservices Architecture Enables DevOps: An Experience Report on Migration to a Cloud Native Architecture, IEEE Software, (2016).
40. J.M Hellerstein, Serverless computing: One step forward, two steps back, arXiv:1812.03651, (2018).
41. Sang GM, et al., A reference architecture for big data systems, The 10th international conference on software, knowledge, information management & applications (SKIMA), Chengdu, China, 15–17, p. 370–5, (December, 2016).
42. Sang GM, et al., Simplifying Big Data Analytics Systems with A Reference Architecture, The 18th IFIP WG 5.5 working conference on virtual enterprises, Vicenza, Italy, 18–20, p. 242–9, (September 2017).
43. Andy S. Alic, et al., BIGSEA: A Big Data analytics platform for public transportation information, Future Generation Computer Systems, Volume 96, Pages 243-269, ( July 2019).
44. Samneet Singh and Yan Liu, A Cloud Service Architecture for Analyzing Big Monitoring Data, Tsinghua Science and Technology, vol. 21, no. 1, pp. 55–70, (Feb 2016).
45. Pekka Paakkonen and Daniel Pakkala, Reference Architecture and Classification of Technologies, Products and Services for Big Data Systems. Big Data Research, Volume 2, Issue 4, Pages 166-186, (December 2015).
46. P. Mcmahon, et al., Requirements for Big Data Adoption for Railway Asset Management, IEEE Access, Volume 8, (January 2020).
47. Dominik Kozjek, et al., Advancing manufacturing systems with big-data analytics: a conceptual framework, International Journal of Computer Integrated Manufacturing, Vol. 33, No. 2, 169–188 , (2020).
48. HPE Reference Architecture for AI on HPE Elastic Platform for Analytics (EPA) with TensorFlow and Spark, Whitepaper, https://www.hpe.com/psnow/doc/a00060456enw, last accessed 2020-05-14.
49. Lui K., Karmiol J., AI Infrastructure Reference Architecture IBM Systems, https://www.ibm.com/downloads/cas/W1JQBNJV, last accessed 2020-05-14.
50. Theo Lynn, et al., A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms, IEEE 9th International Conference on Cloud Computing Technology and Science, (2017).
51. Boubaker Soltani, et al., Towards Distributed Containerized Serverless Architecture in Multi Cloud Environment, The 15th International Conference on Mobile Systems and Pervasive Computing, Procedia, 121–128, (2018).
52. Young bin Kim, et al., Serverless Data Analytics with Flint, IEEE 11th International Conference on Cloud Computing (CLOUD), (2018).
53. Nilton Bila, et al., Leveraging the Serverless Architecture for Securing Linux Containers, IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW), (2017).
54. Kubernetes, https://kubernetes.io/, last accessed 2021/08/05.
55. Josep Sampe, et al., Serverless Data Analytics in the IBM Cloud, Proceedings of the 19th International Middleware Conference Industry, Pages 1–8, (December 2018).
56. Apache MapReduce, https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html, last accessed 2021-08-08.