A Baseline Cloud-based Scalable Serverless Architecture for Batch Processing in Big Data Analytics

Kaustuv Kunal

Abstract. Serverless architectures have become increasingly popular in the software industry, offering attractive benefits such as cost-effectiveness, rapid time-to-market, high reliability and reduced maintenance overhead. These, coupled with the evolving capabilities of public cloud services, have driven a surge in their adoption, particularly in the domain of big data processing. However, constructing such complex systems can pose substantial challenges, particularly for startup organizations with limited expertise. This paper introduces a baseline serverless architecture tailored for scalable end-to-end batch log processing, catering to the data processing requirements of data analytics and mining tasks. The proposed architecture is a four-layer Function-as-a-Service (FaaS) reference model, showcasing scalable and distribution capabilities along with ability to be self-maintained. Furthermore, the architecture is designed to be cost-effective, making it accessible for deployment leveraging any public cloud platform. One of the noteworthy features of this architecture is its comprehensive approach to data management, security, and stakeholder profiling. Moreover, it enhances stakeholder profiling, offering insights into user behaviours, preferences, and other crucial aspects, facilitating better decision-making and personalized user experiences.

The proposed serverless architecture is demonstrated through a case study, showcasing its real-world feasibility and effectiveness. The case study exemplifies the application of the architecture in addressing actual big data processing challenges and highlights its advantages.

By adopting this architecture, organizations can streamline their big data processing pipelines, reducing the complexity and costs associated with traditional infrastructure management. This approach not only accelerates the time-to-market for data-mining solutions but also provides a scalable and efficient framework to handle growing data volumes.

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

1 Introduction

Big data has garnered significant attention in this century, largely driven by the proliferation of the Internet and the exponential growth of digital data. Its impact spans across diverse sectors, including agriculture, commerce, healthcare, and governance. As a consequence, the use of big data has become pervasive, with mobile applications now commonly generating petabytes of data traffic. However, traditional infrastruc-

ture is ill-equipped to handle the vast volume and complexity of big data, necessitating specialized architectures.

These specialized architectures are typically distributed in nature, encompassing crucial stages of data collection, processing, analysis, and visualization. Notably, the advent of distributed architectures such as Hadoop [6] and its ecosystem [7,8,9,10], derived from the Google File System (GFS) [5], gained popularity due to their utilization of cost-effective commodity hardware. However, deploying and maintaining an in-house setup of such infrastructure is often impractical for many enterprises. Fortunately, with the evolution of public clouds, a more viable and efficient option emerged. Public clouds service providers such as, [2,3,4], etc., offered a faster, more flexible, and cost-effective solution for setting up technological infrastructure. This shift significantly reduced the burden of infrastructure management and provided abundant resources and utilities to construct comprehensive processing architectures.

Within the domain of public clouds, the cost considerations are often centered around the balance between compute and storage instances. Serverless architectures [1,11], exemplified by models like Function-as-a-Service (FaaS) and Backend- as-a-Service (BaaS), offer a pay-as-you-go approach, obviating the need for upfront provisioning or server management. These serverless architectures not only reduce infrastructure and operational costs but also offer enhanced scalability and accelerated time-to-market.

The paper specifically focuses on a baseline FaaS batch processing architecture, deliberately designed to leverage any public cloud provider. Notably, one of the challenges lies in designing an effective serverless batch architecture, especially for organizations lacking prior experience in this domain. The paper aims to bridge this knowledge gap and streamline the process, saving valuable time and resources for enterprises. This sequential layered architecture is tailored for mid and small-sized companies, as well as institutions aiming for a cost-effective, scalable, and modular big data batch mining system within the public cloud environment.

To demonstrate the practicality of the proposed baseline architecture, a real-world use case involving ecommerce campaign log analytics is presented. The implementation is showcased on Amazon Web Services (AWS) [2], providing insights into the architecture's applicability and performance.

The structure of the paper is as follows: Related work is discussed in section two, section three provides essential background information, while section four delves into the details of the proposed baseline architecture. In section five, the paper presents the case study, followed by the performance evaluation and limitations in section six. Finally, the paper concludes in section seven, summarizing the key findings and potential avenues for future research.

2 Related Work

To handle the vast and complex data sets, several notable architectures have been proposed, each addressing specific challenges and requirements in the big data domain. One such architecture is the Lambda architecture [35], which seamlessly com-

bines batch and streaming pipelines within a single unified framework. Comprising three key layers, it incorporates a batch layer for distributed processing, a serving layer responsible for incremental indexing of the batch data, and a speed layer designed to complement the serving layer by indexing the most recent data. This architecture efficiently caters to both historical and real-time data processing needs. Another prominent architecture is the Kappa Architecture [36], primarily focused on streaming systems, with the flexibility to extend to historical batch processing. Pipeline [37], Event-driven [38] and microservices architectures [39] are also commonly leveraged for big data processing applications, each providing distinct advantages in handling event-driven data and ensuring modular and scalable solutions. Researchers have proposed big data reference architectures based on the use cases of major tech giants such as Facebook, Amazon, Twitter, and Netflix [42], which culminated in a four-layer abstract reference architecture. Subsequently, this reference architecture was mapped to suit the specific needs of LinkedIn [43]. In [46], a technology-independent reference architecture for big data systems was introduced, drawing insights from published big data use cases and the associated commercial products. Moreover, domain-specific processing architectures and frameworks have been suggested for diverse applications, including public transportation [44], cluster monitoring [45], railway asset management [47], manufacturing [48], and AI [49, 50]. Specifically, in the realm of serverless big data processing, some works have explored serverless architectures on specific public cloud platforms. For instance, in [53], an architecture is presented that utilizes AWS Lambda and Apache Spark libraries to achieve efficient data processing. Similarly, [54] showcases a serverless architecture built on IBM Cloud Functions, tailoring it to the specific requirements of big data tasks and proposes an event-driven serverless architecture based on OpenWhisk [23]and Kubernetes [55], demonstrating the versatility and scalability of serverless solutions.

While the literature review reveals several reference architectures and specific serverless systems for big data, there remains a gap in the availability of concrete and coherent reference architectures in the serverless big data context. This highlights the need for further development in the realm of serverless reference architectures for big data systems. The absence of comprehensive and standardized serverless architectures for big data processing underscores the significance of exploring novel approaches to address the complexities and challenges of managing and processing vast data sets efficiently and cost-effectively.

3 Background

Logging runtime information is a customary procedure, and the widespread integration of applications has subsequently resulted in a significant increase in the amount of log data generated. To handle these massive volumes efficiently and effectively, an intelligent log analytics platform is essential [34]. Logs, due to their high velocity and volume, are considered prime candidates for big data processing, and their timestamped nature makes them valuable for debugging and analysis.

However, log analysis poses several complex challenges:

- Storage: The continuous generation of logs leads to an exponential increase in size, which can result in log server failures due to memory limitations. Managing storage and memory requirements becomes a critical concern.
- Redundancy: A significant portion of log data may lack meaningful business value, and redundant entries can frequently occur. Effectively segregating relevant business data from the vast log pool is a challenging endeavour.
- Structure: The log structure undergoes modification over time due to various factors, such as internal application code changes or external protocol updates. Ensuring seamless adaptation to these log structure modifications poses a substantial challenge.
- Processing: Many logs are readable text data. Due to their non-serialized nature, raw logs are not much suitable for fast processing. Additionally, multiple processing requirements may apply to the same data, and several jobs may necessitate simultaneous access to shared data, requiring efficient data access management.
- Sources: Log data emanates from diverse sources, including sensors, mobile apps, websites, etc. Ensuring the timely collection of logs from heterogeneous sources is a challenging task.
- Stakeholders: Big analytics systems are accessed by multiple stakeholders, warranting the implementation of intelligent access management and user profiling to uphold data security and privacy.

In response to these challenges, cloud computing has emerged as a compelling solution capable of handling the high production rate, large size, and diversity of log files. Companies such as Facebook, Google, Microsoft, Amazon, eBay, etc., have successfully adopted cloud computing for log processing, substantiating log analysis as a viable cloud use case. Among various cloud computing approaches, the rising popularity of serverless big data processing based on FaaS platforms is evident both in industry and research circles. FaaS platforms offer developers the convenience of defining their applications solely through a set of service functions, thereby relieving them from the burdensome task of infrastructure management, which is seamlessly automated by the platform. The multifaceted challenges in log analysis and the benefits of cloud computing, particularly serverless big data processing using FaaS platforms, underscore the relevance and potential of leveraging such architectures for the development of intelligent log analytics systems.

4 The Architecture

This baseline end-to-end cloud-based architecture is crafted for the processing of timestamped log data across diverse domains. Employing a FaaS paradigm, the architecture is structured into four modular layers, each offering distinct entity options. The input to our layered architecture is derived from a log server (or any storage dataset), with the output directed towards analytics or modeling backends. Data is conventionally organized into time-ranged folders. Each layer within the architecture is assigned a specific function, and the associated techniques are elucidated in the initial

part of this section. The architecture leverages three primary cloud components: first, storage units; second, compute instances; and third, messaging channels. These components and their integrations are comprehensively discussed in the latter part of this section.

4.1 The Layers

The architecture comprises four consecutive layers: Fetch, Transform, Process, and Customize, as illustrated in Fig 1. Each layer is designed to function as a server, executing a specialized task and forwarding the processed data to the subsequent layer. These layers can be viewed as independent ETL units, collectively orchestrating the data processing pipeline. Moreover, each layer operates concurrently, processing its respective time unit data parallelly.

Fig. 1. Four sequential FaaS layers

The Fetch layer, In the initial layer, data retrieval from an external log server occurs without any modification. This process involves periodic polling, followed by a straightforward copy operation. Concurrent polling of the last n folders containing timestamped log data (organized as time-labeled folders) is executed to identify any new arrivals. Each log folder signifies a unit of time, such as an hour. The value of n should be sufficiently large to accommodate potential delays and out-of-memory errors. An optimal approach to determining n involves assigning it the value corresponding to the maximum delayed arrival, as represented by equation-(1).

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

Here, l denotes the measured list of data latencies. However, n may assume a large enough value. A more refined approach involves employing box plot boundary values. Equation-(2) presents a quartile-based formula for calculating n, where Q3 and IQR represent the third quartile and interquartile range, respectively.

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

The Fetch layer primarily endows the architecture with independence from external systems, such as log servers, ensuring that the system possesses ample data for debugging, validation requirements, and potential data mismatches. Moreover, a dynamic mechanism involving the recording of data arrival details in a meta file and a scheduled job for periodic computation of quartile and n values is suggested. The

fetched data is stored in a bucket-based cloud storage system, with various storage options explored in the initial part of the subsequent section. Additionally, this section encompasses tasks such as the conversion of heterogeneous logs into a homogeneous format, elimination of duplicate entries, and the provision of pre-ingestion and post-ingestion checks.

The Transform layer's, main objective is to optimize the processing speed of the raw logs. Given that plain row logs tend to be slow to process, transforming them into a machine-readable binary format, a process known as Serialization, can significantly expedite the data processing pipeline. Additionally, other transformation techniques, such as indexing and compression, shall also applied to enhance the extraction and storage efficiency of data.

Numerous data formats are accessible for transformation, and the selection of the appropriate format is contingent upon various factors, including the log properties, processing objectives, and data evolution requirements. Some of the commonly used conversion formats are Parquet [13,14], Avro [15] and ORC [16]. Selection of the appropriate format involves considering factors such as the log structure, processing type, and data fields modification capabilities. For nested structures, formats like Parquet, Avro, or JSON are well-suited. Parquet is preferable for compression-heavy requirements, while Avro excels in handling schema evolution. Columnar formats are recommended for selective processing, such as SQL queries, whereas row formats are more suitable for ETL operations. Additionally, the use of compression techniques, such as Snappy, LZ4, etc., shall be applied on a case-by-case basis in order to further optimize the process. It is important to note that the Transformation layer is optional and can be bypassed if the raw logs are already pre-formatted. Upon transformation of the data, the subsequent step involves data processing.

The Data Processing Layer, serves as the central core of the architecture, taking on the crucial responsibility of executing vital big data processing tasks. In the realm of serverless compute instances, it is important to acknowledge that metastore-based frameworks like Hive are ill-suited. Therefore, data processing occurs within compute units, which leverage specific processing frameworks. This approach ensures efficient resource utilization, with compute instances being occupied solely for the duration of execution. The selection of the appropriate processing framework depends on the specific processing requirements. Among the myriad of available options, frameworks such as Spark [17], Cascading [18], Beam [19], NiFi [20], Flume [56] and Tez [57] are viable candidates. Interested parties can gather in-depth information about each framework from their respective project websites. These frameworks offer support for an extensive array of data processing tasks, encompassing iterative jobs, stream processing, graph processing, and machine learning. Resource optimization is achieved by ensuring that compute instances are utilized only during the execution of processing tasks, thus minimizing idle time and maximizing efficiency. While selecting a processing framework, it is crucial to consider not only the log structure and pro-

cessing requirements (such as mining, retrieval, and prediction) but also the availability of programming resources within the organization.

The Customization Layer, plays an integral role in interacting with external systems. Within this layer, the processed data undergoes further customization to cater to the specific micro requirements of various tasks, such as report generation, modeling, sampling, visualization etc,. The necessity for identical data across various platforms, including databases, visualization tools, analytics systems, and data science modeling frameworks, is a commonplace scenario, albeit with nuanced variations. This layer meets these requirements by generating diverse data variants. Furthermore, the layer facilitates the seamless transfer of data either directly to the frontend or to the respective backend systems, enabling smooth data flow across the architecture. To expedite the setup and tuning processes, specific metadata relevant to each client is attached, ensuring personalized and efficient data processing. Flexibility is a key feature of the Customization Layer, as it allows logic to be implemented using any programming or scripting language. This feature grants users the freedom to adopt languages that best suit their requirements and expertise, thereby enhancing the versatility and adaptability of the architecture.

4.2 The Components

The proposed serverless architecture is depicted in Fig 2. It has three main components: the storage unit (SU), the computation unit (CU), and the messaging queue (MQ). We will examine each of these components one at a time.

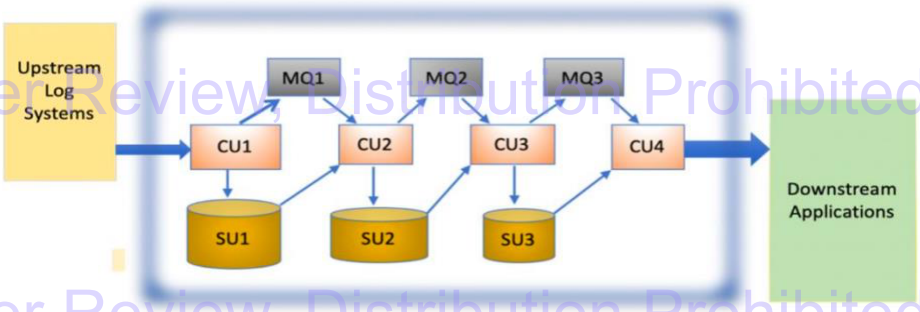

Fig. 2. Serverless Architecture Overview

The Storage Unit (SU), serves as a crucial component in the architecture, responsible for the storage of data across different layers. Each layer, except layer-4 i.e., Customization layer, possesses a dedicated storage unit. In designing these storage units, several desirable characteristics are considered, including robust support for rich APIs, archival capabilities, self-hosting ability, and cost-effectiveness. Bucket-based storage solutions are favoured in serverless systems due to their efficient data organization and management. Data compression support is a significant aspect of these storage units, as once data undergoes transformation, processing, and customization,

the historical data is primarily required only during validation and debugging stages. Regular data compression aids in optimizing storage space contributing to overall system efficiency.

In the context of public cloud environments, various options for storage units are available. Amazon Simple Storage Service (S3), offered by AWS, is an early pioneer in object storage, enabling users to upload vast amounts of data without worrying about storage limitations. It offers provisions for storing both low-latency data (S3-standard) and archival data (S3-Glacier), with seamless data transfer between storage types through lifecycle management mechanisms. Despite its advantages, users need to be mindful of security practices and the choice of regions for optimal performance. GCP Cloud Storage Buckets present a storage and retrieval system supported by Google's reliable and high-speed networking infrastructure. This system efficiently handles data operations while ensuring security and cost-effectiveness. Google Cloud Storage Buckets are classified into four performance tiers: standard, nearline, cold-line, and archive, catering to diverse storage requirements. Azure Blob Storage is designed to store and retrieve highly scalable and unstructured data as Binary Large Objects (BLOBs). It offers various storage tiers, such as hot storage for frequently accessed data, cool storage for less frequently accessed data, and archival storage for data with rare access needs.

Additionally, notable bucket storage options are available from IBM [21] and Oracle [22], each offering specific features and capabilities in the cloud storage landscape. The choice of a suitable storage unit depends on factors such as data access patterns, performance requirements, budget constraints, and the specific needs of the architecture. Selecting the most appropriate storage unit is a crucial decision, as it directly impacts the overall efficiency, scalability, and cost-effectiveness of the serverless architecture.

The Computation Unit, serves as the core of the serverless architecture, executing code after fetching data from storage units. Each layer has an intended compute unit, and its desirable qualities include dynamic memory allocation, auto-scaling, and support for microservice architecture. Noteworthy design candidates encompass:

AWS Elastic Beanstalk, streamlines the provisioning, scaling, and load balancing processes, simplifying the deployment of applications. Its strong support for Docker containers allows for flexible application deployment, and it offers compatibility with various server environments such as Apache HTTP Server, Nginx, Microsoft IIS, and Apache Tomcat.

AWS Lambda, positioned as one of the pioneering Function-as-a- Service (FaaS) platforms on the public cloud, AWS Lambda revolutionizes serverless computing by enabling code execution without the burden of server management. This micro-services environment allows for scalable execution of code for diverse application types and backend services, with the added advantage of cost-effectiveness through its pay-as-you-go model based on compute time consumption. Google App Engine (GAE), emerges as Google Cloud's fully managed serverless application platform, equipped with automatic scaling-up and scaling-down capabilities for applications. It streamlines infrastructure concerns, including patching, server management, load balancers, and built-in mem-cache functionalities, allowing developers to focus solely

on application development and functionality. Google Cloud Functions, a scalable and pay-as-you-go FaaS platform, Google Cloud Functions empowers users with serverless code execution, without the complexities of server provisioning and management. It facilitates automatic scaling based on workload demands and boasts integrated monitoring, logging, and debugging features. Additionally, this platform ensures security at both the role and per function level, adhering to the principle of least privilege. Google Cloud Run - this serverless compute platform stands out as an efficient solution for running stateless containers that can be invoked via HTTP requests. By operating on a fully managed and pay-as-you-use basis, Cloud Run offers a cost-effective approach to computing, as users are only charged for the resources utilized during container execution. Microsoft Azure Functions - positioned as a serverless computing service within the Microsoft Azure public cloud, Azure Functions enables developers to execute small code pieces written in a variety of programming languages, including Node.js, C#, Python, PHP, and Java. This platform allows for on-demand code execution without the need for underlying infrastructure management, simplifying application development and deployment processes. Microsoft Azure App Service: Designed to facilitate rapid development, deployment, and scaling of web apps and APIs, Azure App Service offers a versatile platform to meet diverse application needs.

The choice of the most appropriate computation unit depends on the specific requirements of the application, workload characteristics, and the availability of technical expertise within the organization. Each design candidate presents unique attributes that cater to various serverless computing scenarios, offering researchers and practitioners a diverse array of options to match their specific use cases and performance objectives.

Message Queues, play a crucial role as communication units within the proposed serverless architecture, primarily used for triggering compute units while also maintaining execution state and tracking the status of time-folders. Each time-folder is associated with a specific Message Queue, and these queues enable concurrent processing of messages. The asynchronous nature of communication ensures that each message is processed only once by a single consumer, typically the compute instance, making it a one-to-one or point-to-point communication pattern. The integration of Message Queues offers numerous benefits, including scalability, testability, maintainability, and flexibility to the overall architecture, enhancing its robustness and adaptability to varying workloads.

Several candidates for Message Queue implementations exist, each catering to different use cases and performance requirements. Some notable options are as follows: Apache Kafka [24]; as an open-source event streaming platform, Apache Kafka excels in messaging, stream processing, data integration, and data persistence. It is designed to handle very high throughput, capable of processing thousands of messages per second, making it well-suited for stream processing scenarios. RabbitMQ [25]; known for its lightweight nature and ease of deployment, RabbitMQ is highly adaptable, supporting multiple messaging protocols. It can be deployed in distributed and federated configurations, making it suitable for high-scale and high-availability demands. With broad compatibility across various operating systems and cloud envi-

ronments, RabbitMQ provides an array of developer tools supporting popular programming languages. Amazon SQS i.e., Amazon Simple Queue Service [26] by AWS is a fully managed message queuing service, empowering developers to decouple and scale microservices, distributed systems, and serverless applications. SQS streamlines message-oriented middleware management, allowing developers to concentrate on the distinctive aspects of their work. It offers two types of message queues, Standard and FIFO, each designed to address specific delivery requirements and ordering guarantees. Google Cloud Pub/Sub [27]; this scalable and durable event ingestion and message delivery system enables the creation of infrastructure for handling message queues. Utilizing topics and subscriptions as its core components, Pub/Sub offers low-latency and durable messaging, ensuring all messages sent to a specific topic are delivered to all attached subscriptions. GCP Cloud Tasks [28], represents a fully managed queuing system service for Google Cloud Platform, facilitating the management, dispatch, and delivery of a large number of distributed tasks. It excels in performing work asynchronously outside user or service-to-service requests. Azure Queue Storage [29]; an entity in the Azure cloud infrastructure, Queue Storage allows storage of a vast number of messages accessible worldwide via authenticated HTTP or HTTPS calls. With support for message sizes up to 64 KB and the ability to accommodate millions of messages, it is an ideal choice for creating backlogs of work to process asynchronously. Azure Service Bus [30]; this component forms part of the broader Azure messaging infrastructure, enabling queuing, publish/subscribe, and advanced integration patterns. Designed to integrate applications or application components that span multiple communication protocols, data contracts, trust domains, or network environments, Azure Service Bus offers a comprehensive solution for message handling and communication.

In conjunction with Message Queues, the architectural framework encompasses supplementary elements such as local or remote job invocation nodes (for instance, AWS Elastic Compute Cloud), mechanisms for remote procedure calls (RPC), and Continuous Integration/Continuous Deployment (CI/CD) deployable mechanisms like Jenkins [33]. The heterogeneous assortment of these constituents presents researchers and practitioners with a spectrum of choices to customize the serverless architecture to precise utilization scenarios and objectives pertaining to performance.

5 Case-Study

Introducing a case study that has been executed by leveraging the proposed architectural framework. This serves as both a reference architecture and a means to comprehend the overarching structure, encompassing related functionalities and data flows.

Advertisers seek insights into their campaigns to refine them further. The voluminous data generated from campaigns necessitates processing through big data technologies, especially for analytics and modeling purposes. When a user clicks on an ad, their activities are logged into the campaign log server. The evaluation of a publisher's ad involves various parameters primarily, location, device, operating system, and user

profile. Additionally, predictions are required for metrics like Click-Through Ratio (CTR), Cost-Per-Mile (CPM), Cost-Per-Click (CPC), among others.

Utilizing the proposed architecture, a FaaS based campaign analytics system was developed and deployed on the AWS platform. Each campaign's data is processed in parallel and independently of others. AWS S3 serves as the storage unit, while AWS Lambda acts as the compute unit. Due to our minimal constant queue data requirements, AWS SQS was deemed both suitable and cost-effective. This system contains meta-information, primarily comprising time (year, month, day, hour), layer, publisher, and campaign information. The overall implementation is illustrated in Fig 3. Subsequently, we will delve into the four layers of the system.

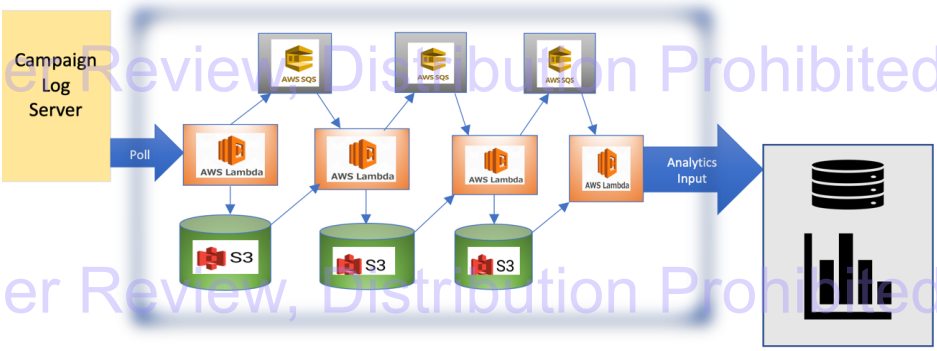

Fig. 3. Architecture Implementation Using AWS

5.1 Fetch Layer: Data Acquisition and Preliminary Processing

The acquisition of real-time data initiates with the depositing of live data into the log server, segregated into discrete hourly data-folders. These campaign log servers, manifested as Amazon Elastic Compute Cloud (EC2) instances, become subjects of a systematic polling mechanism that retrieves newly arrived data for the preceding seven days. This polling is effectuated through a Spring-based REST framework, with the polling operation scheduled to recurrently transpire every hour. Upon detecting new data arrivals, a twofold process ensues. Firstly, the fresh data is copied into an AWS S3 bucket, thereby ensuring data persistence. Simultaneously, a consequential event transpires in the form of the generation of an AWS SQS message. The orchestration of this process transpires within the realms of AWS Lambda, a serverless compute service endowed with the prerogatives of provisioning, scaling, and runtime management. Of particular importance is the metadata contained within the SQS message, a metadata repository that predominantly delineates the data-folder designation and the intended invocation stratum for subsequent processing stages.

5.2 Transform Layer: Data Refinement and Formatting

The transform layer is invoked by SQS messages generated in the fetch layer. The jobs that run in the transform layer are serverless compute instances, specifically AWS Lambda functions. These functions generate the S3 path from the message content and then fetch the plain log data from the path. The transformation process is then initiated.

For data formatting, Avro is the preferred choice. This is mainly due to two reasons. First, Avro has robust support for schema evolution. This means that the schema of the data can change over time without breaking the existing code. Second, Avro is compatible with JSON, which is the format of our majority of log structures. This makes it easy to read and write Avro data.

Since no field preference is specified by the client, all log fields are treated as equally important. This means that no field is dropped or ignored during the transformation process. Columnar storage is not preferred in this case because it would require us to specify which fields are important.

Once the transformation is complete, the Avro files are transferred to a separate transformation S3 bucket. Finally, an SQS message is generated for the transformed time-folder. This message invokes the next layer in the process.

5.3 Process Layer: Data Aggregation and Structured Processing

The workflow in this layer mirrors that of the transform layer, with the input bucket designated as the transform layer's S3 bucket and the output directed to the process layer's bucket. Three primary processing frameworks are employed based on the processing requirements: MapReduce, Cascading, and Spark. Spark, while the fastest, incurs higher costs due to elevated memory demands. MapReduce initially met our requirements at a reasonable cost, while Cascading reduced development cycles with its concise code structure [31]. Microservice architectures like CloudBreak [32] are found to be beneficial for job execution and transferability in this layer. The predominant output data format is .csv files. The jobs primarily involve summarization across various sectors such as location, device model, publisher, operator, referrer, etc., based on temporal factors such as daily, monthly, yearly, etc.

5.4 Customize Layer: Tailoring and Dissemination of Processed Data

Upon the completion of the transformation and processing stages, the customization layer is invoked. Multiple client-facing jobs execute within the serverless compute instances, catering to two main categories of applications. Firstly, an analytics system for visualizing campaign data, where summarized data is stored in MySQL through Java-based jobs utilizing JDBC. Secondly, Data Science (DS) applications for predictive analysis, where DS jobs, written in Python, consume .csv files from S3. This layer is accessible to various developer groups and stakeholders, with multiple data-level access control groups defined to facilitate efficient data access, management,

and security. For code management and version control, Git is employed, while Maven and Jenkins are chosen for build and CI/CD deployment, respectively.

6 Performance & Limitations

More than two dozen live ad campaigns were successfully supported by the deployed system. The largest campaign witnessed the generation of close to a million records per day. Initially, timeout errors and occasional hanging issues were experienced in the serverless compute instances of the process layer for long-running applications. The resolution involved code modifications and the activation of the restartability feature. Subsequently, a checkpoint mechanism was introduced to address potential data loss concerns. Due to non-disclosure agreements with clients, detailed AWS component configurations, specific data processing specifications, and performance evolution details have not been disclosed.

In terms of limitations, a polling strategy is employed to capture delayed data for the last 'n' days, with the default 'n' set at seven (i.e., a week). The processing of delayed files necessitates the reprocessing of entire folders, leading to redundancy and multiple processing instances. This issue is mitigated by the introduction of transaction IDs mapped to each time folder, followed by the removal of older entries in the case of duplicate transaction IDs. However, the impact of this limitation has been minimal, given the relatively modest hourly folder sizes (ranging between 2 to 100 megabytes) and processing times typically not exceeding a few minutes. Consequently, the incorporation of this feature has been made optional.

Additionally, as the architecture is designed for public cloud use, it is advised against its utilization for applications involving classified data, such as those in the banking sector. Furthermore, the system is inherently susceptible to typical drawbacks associated with serverless architectures [41], including the absence of mechanisms for pushing computation closer to data.

7 Conclusion

The paper introduces a cloud-based serverless architecture for analytical mining by leveraging big data log processing techniques & libraries. The proposal explores various components and techniques that shall be leveraged in the layered architecture, offering a comparative analysis. Additionally, a case study demonstrates the implementation of the suggested architecture in ecommerce campaign analytics on AWS. Evidently, the modular architecture, characterized by a FaaS foundation and a four-layer structure, exhibits versatility across a wide spectrum of ETL tasks, yielding harmonized outcomes. Particularly, enterprises seeking a cost-efficient and rapid infrastructure or a preliminary proof-of-concept setup within the public cloud domain stand to benefit from the foundational framework outlined in this study. Noteworthy attributes of this architecture extend beyond the conventional advantages of serverless paradigms, as it not only demonstrates enhanced scalability and self-maintenance capabilities but is also encapsulated within Docker containers. This encapsulation

augments access control mechanisms, profile management, and reinforces data security protocols, underscoring the robustness of the architecture.

In terms of prospective directions, the forthcoming endeavours encompass the formulation of integrated baseline architectures tailored for real-time processing contexts. Furthermore, the paper aims to extend its purview towards accommodating deep learning and natural language processing frameworks within the architecture's overarching framework.

References

1. M Eriksen, “Your Server as a Function”, In Proceedings of the 7th Workshop on Programming Languages and Operating Systems, 5:1--5:7, (2014).
2. Amazon-Web-Services, https://aws.amazon.com, last accessed 2024/01/01.
3. Google Cloud, https://cloud.google.com, last accessed 2024/01/01.
4. Microsoft Azure, https://azure.microsoft.com, last accessed 2024/01/01.
5. Ghemawat Sanjay, et al., The Google File System. Proceedings of the nineteenth ACM symposium on Operating systems principles, Bolton Landing, NY, USA. October 19-22, (2003).
6. Apache Hadoop Project, http://hadoop.apache.org, last accessed 2024/01/01.
7. Apache Hive Project, https://hive.apache.org/, last accessed 2024/01/01.
8. Apache HBase Project, https://hbase.apache.org/, last accessed 2024/01/01.
9. Apache Oozie Project, https://oozie.apache.org/, last accessed 2024/01/01.
10. Apache Sqoop Project, https://sqoop.apache.org/, last accessed 2024/01/01.
11. Baldini, et al. “Serverless computing: Current trends and open problems,” in 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. Proc. VLDB Endow., 3:330--339, Sept (2010).
14. Parquet Project, https://parquet.apache.org, last accessed 2024/01/01.
15. Apache Avro Project, https://avro.apache.org,last accessed 2024/01/01.
16. ORC project, https://orc.apache.org/,last accessed 2024/01/01.
17. Spark Project, https://spark.apache.org, last accessed 2024/01/01.
18. Cascading, https://www.cascading.org, last accessed 2024/01/01.
19. Apache Beam, https://beam.apache.org/, last accessed 2024/01/01.
20. Apache Nifi Project, https://nifi.apache.org/,last accessed 2024/01/01.
21. IBM Object Storage, https://www.ibm.com/products/cloud-object-storage, last accessed 2024/01/01.
22. Oracle Object Storage, https://docs.oracle.com/en-us/iaas/Content/Object/Concepts/objectstorageoverview.htm, last accessed 2024/01/01.
23. Apache OpenWhisk Project, https://openwhisk.apache.org, last accessed 2024/01/01.
24. Kafka Project, https://kafka.apache.org, last accessed 2024/01/01.
25. RabbitMQ, https://www.rabbitmq.com, last accessed 2024/01/01.
26. Amazon-SQS, https://aws.amazon.com/sqs, last accessed 2024/01/01.
27. GCP- Pub/Sub, https://cloud.google.com/pubsub, last accessed 2024/01/01.
28. GSP- Tasks, https://cloud.google.com/tasks, last accessed 2024/01/01.

29. Azure - Storage Queues, https://azure.microsoft.com/en-in/services/storage/queues, last accessed 2024/01/01.
30. Azure - Service Bus Messaging, https://docs.microsoft.com/en-us/azure/service-bus-messaging/, last accessed 2024/01/01.
31. K Kunal “Analysing Cascading over MapReduce”, proceedings of International journal of Computer Applications (IJCA), 9(1): 1–5, November, last accessed 2024/01/01.
32. CloudBreak Project, https://docs.cloudera.com/HDPDocuments/Cloudbreak, last accessed 2024/01/01.
33. Jenkins, https://www.jenkins.io, last accessed 2024/01/01.
34. Z Zheng, Z Lan, BH Park, A Geist , System log pre-processing to improve failure prediction, IEEE/IFIP International Conference on Dependable Systems & Networks, (2009).
35. Big Data Principles and Best Practices of Scalable Realtime Data Systems by Nathan Mar, (2015).
36. Kappa Architecture, http://milinda.pathirage.org/kappa-architecture.com, last accessed 2024/01/01.
37. Pipeline Architecture, CV Ramamoorthy, HF Li, ACM Computing Surveys (CSUR), (1977)
38. Event-Driven Architecture Overview By Brenda M. Michelson, Elemental Links, Patricia Seybold Group, (2006).
39. Microservices Architecture Enables DevOps: An Experience Report on Migration to a CloudNative Architecture A Balalaie, A Heydarnoori, P Jamshidi - IEEE Software, (2016).
40. Serverless computing: Current trends and open problems, Baldini, I.; Castro, et al.. In Research Advances in Cloud Computing; Springer: Berlin/Heidelberg, Germany, (2017).
41. Serverless computing: One step forward, two steps back. Hellerstein, J.M.; et. arXiv:1812.03651, (2018).
42. A reference architecture for big data systems, Sang GM, et al., The 10th international conference on software, knowledge, information management & applications (SKIMA), Chengdu, China, 15–17, p. 370–5, December, (2016).
43. Simplifying Big Data Analytics Systems with A Reference Architecture, Sang GM, et al.., The 18th IFIP WG 5.5 working conference on virtual enterprises, Vicenza, Italy, 18–20, p. 242–9, September, (2017).
44. BIGSEA: A Big Data analytics platform for public transportation information, Andy S.Alic, et.al., Future Generation Computer Systems, Volume 96, Pages 243-269, July (2019).
45. A Cloud Service Architecture for Analyzing Big Monitoring Data, Samneet Singh and Yan Liu, Tsinghua Sci. Technol., vol. 21, no. 1, pp. 55–70,Feb. (2016).
46. Reference Architecture and Classification of Technologies, Products and Services for Big Data Systems. Pekka Paakkonen, Daniel Pakkala, Big Data Research, Volume 2, Issue 4,Pages 166-186, December (2015).
47. Requirements for Big Data Adoption for Railway Asset Management, P. Mcmahon et.al, IEEE Access, VOLUME 8, (2020).
48. Advancing manufacturing systems with big-data analytics: A conceptual framework, Dominik Kozjek et.al, International Journal of Computer Integrated Manufacturing, Vol. 33, NO. 2, 169–188, (2022).
49. HPE Reference Architecture for AI on HPE Elastic Platform for Analytics (EPA) with Tensor Flow and Spark, Whitepaper, HPE, (2018).
50. Lui K., Karmiol J. AI Infrastructure Reference Architecture IBM Systems, 87016787USEN-00. https://www.ibm.com/downloads/cas/W1JQBNJV, 2023.

51. A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms, Theo Lynn, et.al, IEEE 9th International Conference on Cloud Computing Technology and Science; (2017).
52. Towards Distributed Containerized Serverless Architecture in Multi Towards Distributed Containerized Serverless Architecture in Multi Cloud Environment, Boubaker Soltani et.al., The 15th International Conference on Mobile Systems and Pervasive Computing, Procedia Computer Science 134, 121–128, (2018).
53. Serverless Data Analytics with Flint, Youngbin Kim, et.al, IEEE 11th International Conference on Cloud Computing (CLOUD),(2018).
54. Leveraging the Serverless Architecture for Securing Linux Containers, Nilton Bila, et. al., IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW), June (2017).
55. Kubernetes, https://kubernetes.io, last accessed 2024/01/01.
56. Apache Flume, https://flume.apache.org, last accessed 2024/01/01.
57. Apache TEZ, https://tez.apache.org,last accessed 2024/01/01.