Serverless Sorting of Large Files: An ECG Use Case

D. Todorov[*,**] and M. Gusev[*]
[*] Sts Cyril and Methodius University in Skopje, Faculty of Computer Science and Engineering, Skopje, North Macedonia
[**] Innovation Dooel, Skopje, North Macedonia
E-mail: david.todorov@innovation.com.mk, marjan.gushev@finki.ukim.mk

***Abstract*—This paper aims at formatting, sorting and processing large files with the use of serverless cloud computing. Each file presents a set of ECG data samples, usually written as a set of textual lines of Unix-based date-time stamp and ECG sample value. Data might come in random order, due to the functionality of the system. Whenever there is a communication between the sensor and the cloud system, the system prefers to use the latest data first, so this is the source of out-of-order file receipt. *Methodology*: Several experiments are conducted using different cloud platforms with goal to figure out the best serverless solution. The system architecture is based on activating a cloud function by a trigger on every new received file. The cloud function needs to process the received file, merge it with an existing large file that stores data from previous files within a given time frame, sort it with as much efficiency as possible, and then continue with further computations that require an analysis of a large time frame. *Data analysis*: We will measure the time our solution takes across multiple serverless computing platforms, analyze the pricing of each, compare and evaluate to figure out which cloud platform best suits the needs of this particular use case. *Conclusion*: In addition to the different cloud platforms, the files can be accessed and processed with different methods using the corresponding serverless tools. We point out the implementation details along with a discussion of benefits and potential problems in their implementation.**

Keywords*— *Serverless; ECG; cloud computing; AWS; Azure; GCP;

I. Introduction

The high costs of buying and maintaining large numbers of machines, and the fact that each was often underutilized, led to a great leap forward: virtualization. Common to both hardware-based and container-based virtualization is the central notion of a server. Servers have long been used to back online applications, but they are notoriously difficult to configure and manage, and server startup time severely limits an application's ability to quickly scale up and down [1].

As a result, a new model, called serverless computation, is poised to transform the construction of modern scalable applications. Serverless computing is a paradigm that enables abstraction of details of server and execution environment configuration, and helps the programmer focus on just the code and business rules of the problems he is solving. A perk that serverless computing vendors offer is a pay-as-you go model, which means that the developer only pays for what is being used in a certain time. If a serverless application is idle, the developer doesn't pay anything, which is not the case for standard hosting with virtual machines or containers, where the price is constant.

If we look at serverless through the prism of developer control, we observe that it sits between two extremes: IaaS (Infrastructure as a service) [2] and PaaS (Platform as a service) [3] or SaaS (Software as a service) [4]. The IaaS model is where the developer has the most control over both the application code and operating infrastructure, whereas the PaaS and SaaS models are where the developer is unaware of any infrastructure, and consequently no longer has control over the infrastructure. Instead, the developer has access to prepackaged components or full applications. Whatever code the developer does host here, it is tightly coupled to the platform. In serverless computing, the developer has control over the code they deploy into the Cloud, though that code has to be written in the form of stateless functions. The developer does not worry about the operational aspects of deployment and maintenance of that code and expects it to be fault-tolerant and auto-scaling [5].

Having a basic grasp of what serverless computing means, we next need to set the groundwork for the actual problem we will be using serverless computing to solve. The system that this solution is a part of is a system that collects ECG heartbeat data from a sensor attatched to a patient's body. That sensor measures the electrical activity of the heart through time and sends the data to a server, which in turn sends it to the system that we are describing. The data is then processed by machine learning algorithms to produce useful results. The problem is, those algorithms need the data to be sorted and formatted properly, and the data which comes in as input is the opposite of that. It is our job to format the data and sort it so that the heartbeat data will be in order and make sense to the algorithms that need to process it.

The files represent individual measurement and can have different sizes, from a few megabytes, to more than a gigabyte. Due to Internet connectivity and network throughput, they don't necessarily come in at consistent time intervals. This plays into the strengths of serverless computing. Namely, we pay for only the resources we use, so if at a given time, a function is processing a 3 MB file, we pay for only those 3 MBs and the time it takes to process them. If the file is 500 MBs, we pay for 500 MBs and the time it takes for that file. If no files come in and the function is idle, that time requires no payment. This is not the case with standard application hosting through VMs or containers.

Serverless computing solutions are provided by all the major cloud computing vendors, the main ones being Amazon with its Amazon Web Servies [6], Google with its Google Cloud Platform [7] and Microsoft with Azure [8]. Our research will try to answer which vendor's services best suit our solution in terms of speed and price.

Two main tools provided by the major cloud service providers will be used for our serverless solution of the problem. First, since we are processing data, we will need a way to store our data somewhere persistently. One of the major cloud services that is provided is cloud storage, which enables the user to upload and store files in the cloud [9]. This can be AWS's Amazon Simple Storage Service (S3), Microsoft's Azure Blob Storage, or Google's GCP Cloud Storage. One important thing to remember about Cloud Storage Buckets is that the objects inside it are not editable like files are on a standard file system. Once a file is written and finalized in a bucket, it is immutable. So if we need to change something, a new file should be created with the required content, the old file should be deleted and the new file should be renamed to the old file's name.

Second, we will need to run some code to process the data we have stored. Cloud service providers offer a serverless computing solution known as FaaS (Function as a Service). Basically, the developer focuses on writing the code and needs not worry about setting up an environment where the code will run, like a virtual machine, or a docker container. The developer writes small, bite-sized snippets of code, sends them to the provider, and the provider does the rest, ensuring the proper parameters are satisfied for the code to be able to run. After a function is deployed, it remains idle until it's triggered by an event. These events can be anything from standard HTTP invocation, to the publication of message to a specific topic that the function subscribes to, or the writing of an object to Cloud Storage. The first time a function is triggered, the cloud service provider needs to do some background work, allocating the needed resources, before the code can actually run. This is why the first time a cloud function is triggered, more time needs to pass until a function is executed, than with following triggerings.

Before the solution is described in more detail, Section II presents other work related to our experiment and/or has a similar use case. Then, Section III provides a close examination of the theory behind our solution and a description of its implementation. The conducted experiments, which the conclusion of the paper depend on, are described in Section IV. The results from the experiments are presented in Section V for each of the cloud vendors. Other approaches and differing strategies for implementing the same solution will be briefly described in Section VI and finally, concluding remarks are given in Section VII.

II. Related work

Since serverless computing is a relatively young paradigm, the research space is not filled in the same way as with other more established concepts. Even less frequent are research papers that deal with big data and large file processing.

Sánchez-Artigas et al. [10] present Primula, a serverless sort operator that abstracts away users from the complexities of resource provisioning, skewed data and stragglers, yielding the most accessible sort primitive to date.

Similar experiments to what we here are trying to accomplish, comes by Pogiatzis and Samakovitis [11] presenting an event-driven Extract, Transform, and Load (ETL) pipeline serverless architecture and providing an evaluation of its performance over a range of dataflow tasks of varying frequency, velocity, and payload size. They design an experiment while using generated tabular data throughout varying data volumes, event frequencies, and processing power in order to measure: (i) the consistency of pipeline executions; (ii) reliability on data delivery; (iii) maximum payload size per pipeline; and, (iv) economic scalability (cost of chargeable tasks).

Some less optimistic views are presented in [12] which starts by analyzing three fundamental trade-offs of today's serverless computing model and their relationship with data analytics. It studies how by relaxing disaggregation, isolation, and simple scheduling, it is possible to increase the overall computing performance, but at the expense of essential aspects of the model such as elasticity, security, or sub-second activations, respectively. The consequence of these trade-offs is that analytics applications may well end up embracing hybrid systems composed of serverless and serverful components, which they call Servermix.

III. Solution Overview

The problem that we need to solve is of an ETL nature (Extract, Transform, Load) [13]. Its solution will be executed in a sort of pipeline, where files come in at the input, are processed, stored somewhere else. Those processed files are then processed further and stored somewhere else, and so on, until the required format is reached. The files are in the comma separated value (CSV) form that come in out of order and are not formatted properly. The stored data is time series data with a UNIX timestamp ts and an ECG value x, separated by comma. Every file contains lines with values of ts_i,x_i, with a formatting example as follows:

$ts_1, x_1, ts_3, x_3, ts_7, x_7, \ldots$

$ts_4, x_4, ts_5, x_5, ts_6, x_6, ts_2, x_2 \ldots$

The first sort of processing that needs to be done is to format them properly where each timestamp,x value is in its own row, like so:

ts_1, x_1

ts_2, x_2

ts_3, x_3

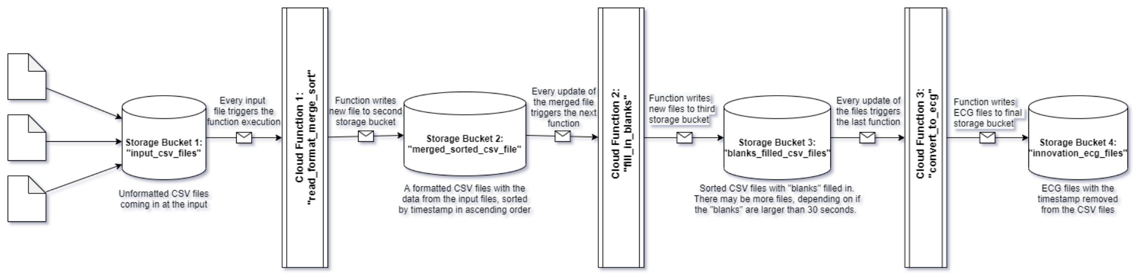
Storage Bucket 1: "input_csv_files"
Unformatted CSV files coming in at the input
Every input file triggers the function execution
Cloud Function 1: "read_format_merge_sort"
Function writes new file to second storage bucket
Storage Bucket 2: "merged_sorted_csv_file"
A formatted CSV files with the data from the input files, sorted by timestamp in ascending order
Every update of the merged file triggers the next function
Cloud Function 2: "fill_in_blanks"
Function writes new files to third storage bucket
Storage Bucket 3: "blanks_filled_csv_files"
Sorted CSV files with "blanks" filled in. There may be more files, depending on if the "blanks" are larger than 30 seconds.
Every update of the files triggers the last function
Cloud Function 3: "convert_to_ecg"
Function writes ECG files to final storage bucket
Storage Bucket 4: "innovation_ecg_files"
ECG files with the timestamp removed from the CSV files

Fig. 1: Graphical representation of the pipeline architecture used for solving the ETL problem

ts_4, x_4

...

The second transformation that needs to be performed is a scaling and translation change to the x value into a y value as specified in (1), assuring that $0 \leq y \leq 1023$.

$$y = \frac{x}{6} + 511 \tag{1}$$

After data formatting according to (1), all of it needs to be collected from all files at the input and merged into one big file, that will likely have a size of several GB. That large file will then need to be sorted by the timestamp so all rows are in ascending order ensuring the time series makes sense.

The key to the next transformation is understanding the data. The timestamps all have a difference of 8 milliseconds between them, but since it relies on real time processing, it can happen that some data hasn't reached the system, or hasn't been generated at all. This means that in the final sorted file, there can be consecutive rows where the timestamps have more than 8 milliseconds of difference between them. We will call these differences of more than 8 milliseconds ”blanks” in the further text. It is our job to fill in the missing timestamp,y pairs with a y value of -1 for each missing row.

If there happens to be a pair of consecutive rows with a blank of 30 seconds or more, the file needs to be split in two. The first file will end at the first row of the consecutive pair, and the next file will begin at the second row of the consecutive pair. The file can be split many times, depending on how many blanks of 30 seconds or more were in the sorted file.

Something that is also a possibility due to the nature of out-of-order inputs, is that the current input files do not contain the complete information, but the next input file that comes in, might fill in some of the blanks that might have been present previously. This means that with every new input file, the merged, sorted and blanks-filled-in files need to be re-computed.

Finally after all previous transformations are done and we have several sorted files, those files need to be converted from csv to an ecg format. We do this by removing the timestamp value from each timestamp,x row, leaving only the x value, and then changing the extension of the file from .csv to .ecg.

A. *Implementation Details*

Two major services provided by cloud vendors are used in our implementation. We are using serverless cloud functions in tandem with cloud storage buckets.

Fig. 1 shows the pipeline architecture that being used to Extract, Transform and Load the data into our desired format. First, a Cloud Storage Bucket needs to be created to store the input files coming in from the outside world. Every time a file is inserted into this bucket, an event is created that triggers the first cloud function.

This first cloud function, named read-format-merge-store, has quite a self-explanatory name. It takes note of the file that generated the event and reads it line by line. Since each line has multiple ”timestamp,x” pairs, it needs to be split, and every pair is added into a list. That list is then sorted by timestamp and is ready to be inserted into the merged file in the next storage bucket. The first time a file is read and the function is executed, the list is written line by line directly into a file created in the output bucket. Every other time, after the list is sorted, a merge-sort is performed with the merged file already present in the output bucket. The merged file is read line by line, and the list of sorted lines is read simultaneously. A temporary file is created so it can store the new content. Lines from each file are compared and the one with a smaller timestamp value is inserted into the temporary file. After one of the sources is exhausted, either the list containing sorted lines from the input file, or the merged file being read line by line, the rest of the lines from the remaining source are written to the temporary file. Then, the merged file is deleted and the temporary file is renamed to take the name of the merged file. This approach, essentially allows us to format, merge and sort the input files all at once.

The next function is triggered by a ”create” event on the storage bucket containing the merged file. The goal of this function is to fill in the blanks of more than 8 milliseconds. If a blank of 30 seconds or more arises, the lines up to that point are stored in one file, and the lines after the blank are stored in another file. The function starts reading from the merged file line by line and always keeps two lines from the file, previous and current, in each iteration. The difference between the timestamps is computed, and if it's 8 milliseconds, the previous line is written to the output file and the current line is made to be previous, while a new line is read from the file. If the difference between timestamps is more than 8 milliseconds, two options are possible. The first is that the difference is 30,000 milliseconds or more, and in this case, the previous line is written to the file and the file writing stream for the file is closed. A new file is opened and the current line is written to it. The second option is that the difference is less than 30,000 milliseconds. In this case, after the previous timestamp is written to the file, new lines are created in between the previous and current timestamp, where the timestamps of the new lines are each with 8 milliseconds of difference between them, and the x values are set to -1. From here, the process continues with reading the next line. After everything is finished, the third storage bucket should have one or more files, sorted by timestamp, where there are no missing timestamps, and each of the lines that were added after the fact, have a -1 as an x value.

The final function is triggered by a ”create” event on the bucket containing the files with blanks filled in. Its job is to convert those files into an ECG file format. It does this by reading a file that triggered the function line by line, removing the timestamp value from each line, keeping only the X value, and writing it in a new file into the next bucket which keeps the ECG files. The new files also need to have the file extension ”.ecg”. With that, the transformation of an input file is complete and the input files bucket awaits the arrival of new files.

As one might have noticed, each function reads from one bucket, but writes to another bucket. This is because the functions are triggered by events on these buckets, and if a function writes to the same bucket it was triggered by, it will be stuck in an infinite loop.

IV. Experiments

The information we are interested in is how this solution scales when the number of files increases and the merged file grows larger. To that effect, several experiments will be conducted, but first, some ground rules need to be established.

The main thing to understand, at least with this specific solution, is that a cloud function that streams data to and from cloud storage buckets, can not be called while it is still being executed. Meaning if the function is in the middle of streaming files to an output storage bucket, and another file comes at the input bucket which triggers said function, two instances of the function would be in contention to write to the same resource in the output bucket, which introduces problems for the integrity of the data. This would mandate that the files at the input bucket be inserted one at a time, with a certain time interval between each insertion so as not to interrupt any currently running function. To that end, the experiments are conceived so that the execution for the Nth invocation of the function will be measured. If we N is set to be 10, that would mean 9 files are first inserted one by one to the input bucket, and on the 10th file, the time to complete the function execution will be measured.

TABLE I: Execution time in seconds for different numbers of files - Google Cloud Platform

No. of files	read-format-merge-sort	fill-blanks	convert-to-ecg
5	9.8	14.39	5.38
10	19.52	28.88	10.41
20	32.92	52.74	21.89
30	40.28	74.55	32.12
50	47.89	117.26	54.23
75	53.47	165.38	79.65
100	57.96	211.84	105.04

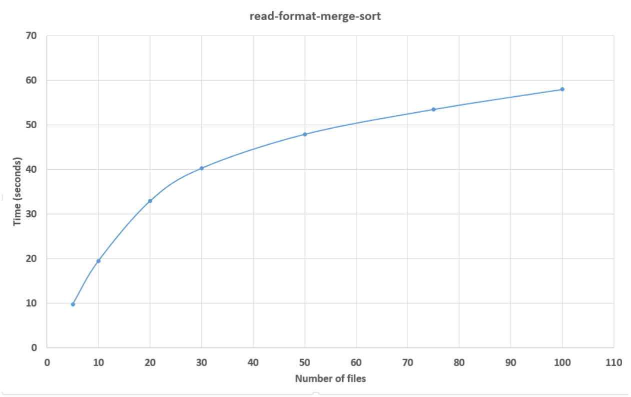

Fig. 2: Execution time for read-format-merge-sort

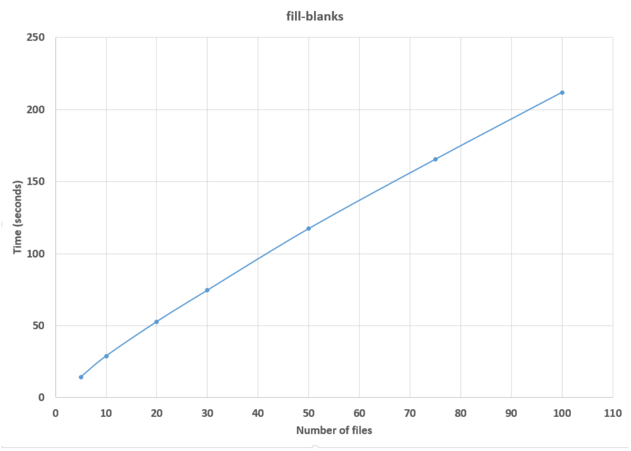

Fig. 3: Execution time for fill-blanks

Now that the rules are clear, the testing of the pipeline will be conducted with a variety of values for N, ranging from 5 to 100. The results will then be compared for different cloud service providers to determine which one offers the best service for the price.

V. Results

Results were gathered by insterting the files into the input bucket one by one and waiting to see change in the final output bucket before inserting a new file. As mentioned before, this is because an instance of a function fails if it tries to use a resource already being used by another instance of a function.

The results gathered for the experiment using N number of files, for N between 5 and 100, are shown in Table I,

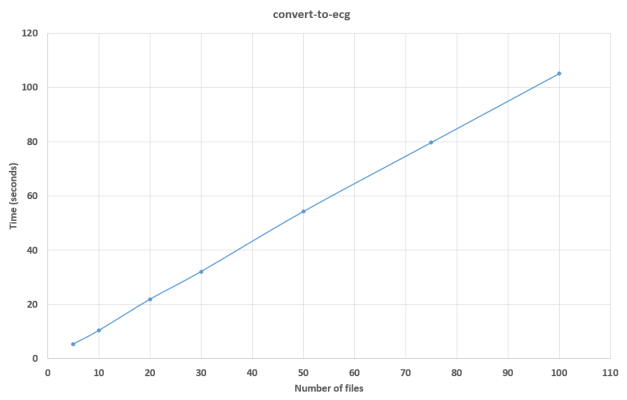

Fig. 4: Execution time for convert-to-ecg

TABLE II: Execution time in seconds for different numbers of files - Amazon Web Services

No. of files	read-format-merge-sort	fill-blanks	convert-to-ecg
5	8.63	12.97	3.79
10	19.03	27.65	9.15
20	31.62	52.42	20.31
30	41.34	76.37	31.56
50	47.28	120.11	53.74
75	54.16	169.42	83.22
100	60.17	213.73	108.68

Table II and Table III for Google Cloud Platform, Amazon Web Services and Microsoft Azure respectively.

Figure 2, Figure 3 and Figure 4 show the relationship between the execution time and the number of files for the functions read-format-merge-sort, fill-blanks and convert-to-ecg, respectively.

It can be observed that the relationship between number of files and execution time is linear for the fill-blanks and convert-to-ecg functions, while it's logarithmic for the read-format-merge-sort function. This is because the former only iterate the files, occasionally performing insertions/deletions, which amounts to a O(n) complexity, while the latter uses part of a merge sort algorithm, namely the merge, which has a complexity of O(log(n)).

Graphs for AWS and Azure weren't drawn because the numbers are almost identical to the GCP numbers, within a margin of error of about 5 seconds.

VI. Discussion

The striking observation of these experiments isn't the difference between the cloud vendors, because there really isn't much of one. The results were variable, to say the least and that is due to current operating system load, network load, which can not be controlled.

TABLE III: Execution time in seconds for different numbers of files - Microsoft Azure

No. of files	read-format-merge-sort	fill-blanks	convert-to-ecg
5	11.6	16.48	8.13
10	20.48	31.27	12.67
20	32.54	51.34	24.09
30	39.72	74.98	31.86
50	46.02	115.03	52.75
75	51.24	159.46	77.47
100	55.63	206.52	103.28

What is striking is the observation of how unscalable this solution seems to be, in light of these results. 100 files collectively containing less than 100 MB is still small compared to the 1.5 GB of data that will need to be processed in a regular use case.

The slow performance of the pipeline can mostly be attributed to one main contributing factor, and that is the nature of cloud storage. Objects stored in cloud storage are inherently immutable, which means they cannot be changed once they have been finalized, which isn't the case with files in a standard file system. The main reason this has an effect on speed is because whenever we want to add a new line to an existing file, essentially we need to create a new file, write all of the content from the original file into the new one, then add the new line to the new file, and finally delete the old file. There isn't a standard way to append content to a file like there is with a standard file system. The solution is, thus, built around this limitation, and the outcome reflects that limitation.

VII. Conclusion

The advantages of serverless computing are plentiful, from the fact that development time is drastically reduced, abstracting the details of infrastructure management and placing all the focus on the actual business logic; to the decrease in price, depending on the user base and usage. Not every problem, however, is suited to the use of the serverless paradigm.

The pricing model of serverless computing, combined with the nature of immutable objects in cloud storage, a crucial service to be utilized if a serverless approach is to be taken, and the picture being painted is that this problem is not suited for a serverless solution.

Serverless computing is best used for processes that have a short execution time, or processes that depend mostly on user interaction. What serverless is not suitable for are processes that require a long time to finish. Since serverless computing providers charge for the amount of time code is running, it could become more expensive to use a serverless infrastructure to host our code, than a traditional one.

References

[1] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with OpenLambda," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association, Jun. 2016. [Online]. Available: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson

[2] R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente, "Iaas cloud architecture: From virtualized datacenters to federated cloud infrastructures," *Computer*, vol. 45, no. 12, pp. 65–72, 2012.

[3] E. Keller and J. Rexford, "The" platform as a service" model for networking." *INM/WREN*, vol. 10, pp. 95–108, 2010.

[4] A. Dubey and D. Wagle, "Delivering software as a service," *The McKinsey Quarterly*, vol. 6, no. 2007, p. 2007, 2007.

[5] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, “Serverless computing: Current trends and open problems,” 2017.

[6] E. Amazon, “Amazon web services,” *Available in: http://aws. amazon. com/es/ec2/(November 2012)*, p. 39, 2015.

[7] E. Bisong, “An overview of google cloud platform services,” *Building Machine Learning and Deep Learning Models on Google Cloud Platform*, pp. 7–10, 2019.

[8] M. Copeland, J. Soh, A. Puca, M. Manning, and D. Gollob, “Microsoft azure,” *New York, NY, USA:: Apress*, 2015.

[9] J. Wu, L. Ping, X. Ge, Y. Wang, and J. Fu, “Cloud storage as the infrastructure of cloud computing,” in *2010 International conference on intelligent computing and cognitive informatics*. IEEE, 2010, pp. 380–383.

[10] M. Sánchez-Artigas, G. T. Eizaguirre, G. Vernik, L. Stuart, and P. García-López, “Primula: A practical shuffle/sort operator for serverless computing,” in *Proceedings of the 21st International Middleware Conference Industrial Track*, 2020, pp. 31–37.

[11] A. Pogiatzis and G. Samakovitis, “An event-driven serverless etl pipeline on aws,” *Applied Sciences*, vol. 11, no. 1, p. 191, 2021.

[12] P. G. López, M. S. Artigas, S. Shillaker, P. R. Pietzuch, D. Breitgand, G. Vernik, P. Sutra, T. Tarrant, and A. J. Ferrer, “Server-mix: Tradeoffs and challenges of serverless data analytics,” *CoRR*, 2019.

[13] P. Vassiliadis, A. Simitsis, and S. Skiadopoulos, “Conceptual modeling for etl processes,” in *Proceedings of the 5th ACM international workshop on Data Warehousing and OLAP*, 2002, pp. 14–21.