applied sciences

MDPI

Article

An Event-Driven Serverless ETL Pipeline on AWS

Antreas Pogiatzis * and Georgios Samakovitis *

School of Computing and Mathematical Sciences, University of Greenwich, Old Royal Naval College, Park Row, Greenwich, London SE10 9LS, UK
* Correspondence: a.pogiatzis@greenwich.ac.uk (A.P.); g.samakovitis@greenwich.ac.uk (G.S.)

Abstract: This work presents an event-driven Extract, Transform, and Load (ETL) pipeline serverless architecture and provides an evaluation of its performance over a range of dataflow tasks of varying frequency, velocity, and payload size. We 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). We run 92 parameterised experiments on a simple AWS architecture, thus avoiding any AWS-enhanced platform features, in order to allow for unbiased assessment of our model's performance. Our results indicate that our reference architecture can achieve time-consistent data processing of event payloads of more than 100 MB, with a throughput of 750 KB/s across four event frequencies. It is also observed that, although the utilisation of an SQS queue for data transfer enables easy concurrency control and data slicing, it becomes a bottleneck on large sized event payloads. Finally, we develop and discuss a candidate pricing model for our reference architecture usage.

Keywords: serverless; FaaS; event-driven; distributed; AWS; ETL; architecture

check for updates

Citation: Pogiatzis, A.; Samakovitis, G. An Event-Driven Serverless ETL Pipeline on AWS. *Appl. Sci.* **2021**, *11*, 191. https://dx.doi.org/10.3390/app11010191

Received: 31 October 2020
Accepted: 22 December 2020
Published: 28 December 2020

1. Introduction

Efficient, scalable, and cost-effective data processing and pipelining have become critically important in real-time analytics for decision making. Naturally, the paradigm favours real time event-driven solutions over periodic batch processing. Systematic techniques for these tasks have been thoroughly investigated and many open source tools and frameworks emerged and adopted by industry [1–4]. Even so, these are predominantly distributed solutions that require costly infrastructure and significant effort for development and maintenance.

Data processing requirements in such environments most frequently consist of at least one typical Extract, Transform, and Load (ETL) pipeline in order to accommodate for heterogeneous sources. Generally, an ETL pipeline entails a data source that receives incoming data, a computation stage that is scheduled to run at fixed intervals and a data sink to store the post-processed data. Event-driven ETLs offer an alternative approach, removing the need for fixed interval runs by operating in a more reactive manner, by allowing changes in the data source to trigger data processing. This approach features real time feedback, the efficient utilization of resources and elasticity [5], and it is often more desirable with respect to business requirements. Yet, it is intrinsically harder to implement due to its architectural complexities.

Recent advancements in serverless computing afford new opportunities in designing architectures that realise high scalability, elasticity, and performance, while minimising the cost and development effort. Such advancements have lowered entry barriers in the implementation of event-driven ETL pipelines. In this paper, we adopt an event-driven ETL pipeline architecture that was built on the AWS platform while using exclusively serverless technologies and offer experimental evidence for its performance. To do so, we develop a reference implementation of an event driven serverless architecture that is based on various serverless design patterns [6–10].

An empirical evaluation is performed through a series of experiments across different dimensions in order to test for consistency, reliability, pricing, performance, and payload limits of the architecture. In this context, this research

- Presents a reference architecture of a serverless ETL pipeline on AWS and indicates advantages and limitations.
- Provides a data-driven analysis of the architecture through 92 distinct experiments.
- Conducts a critical analysis on the results, evaluates the critical aspects of the system, discusses bottlenecks, and suggests alternative solutions.

Because one of the key motivations of this empirical research is to explore how the reference serverless architecture behaves under a variety of parameters, we believe this work offers a solid contribution to that end. It is imperative to clarify that the scope of this research does not entail a comparative evaluation with other ETL solutions. Conversely, it proposes a serverless architecture in order to accommodate ETL pipelines and empirically demonstrates its capabilities and limitations.

2. Background

Serverless computing is a relatively recent and increasingly popular evolution of cloud computing technology. It aims to provide a new programming model that fully abstracts away the infrastructure layer for developers [11]. Throughout the literature, the term "serverless" most often refers directly to Function-as-a-Service (FaaS) providers; however, the term FaaS does not cover all serverless technologies. Figure 1 provides a view of the spectrum of services that are typically classified as serverless. In this paper, the term "serverless" refers to the wider definition that wraps these technologies.

Recent major industry shifts towards containers and microservices [12] led, in turn, to increasing interest towards serverless computing. A main factor of attraction is that the serverless paradigm supports a pay-as-you go model, allowing for (i) cost-effective agile development and rapid prototyping [13] and (ii) acclaimed elasticity from zero to "infinity", given that scaling, parallelisation, and concurrency are delegated to the cloud provider [14]. Therefore, it becomes significantly easier for developers to roll-out cloud-based deployments that fulfil demanding non-functional requirements in a cost effective manner, avoiding the labour of managing the infrastructure. As already mentioned, although the scope of this work is not to formally compare this system with the traditional ETL solutions, there are also intrinsic serverless advantages against serverful approaches that are inherited by the proposed architecture. Table 1 summarises a high level comparison of traditional and serverless ETL solutions across the aforementioned aspects.

Despite these new opportunities, serverless computing also poses significant risks from the user perspective. Adopting serverless architectures without proper planning can potentially hinder the smooth and successful development process. It requires rigorous design and planning to ensure: (i) architecture parity with Quality of Service (QoS) requirements and (ii) controllable costs as deployment scales [14]. A recent study identified use cases, where FaaS services fail to meet expectations mostly due to design-driven constraints [15]. It is important to understand that shifting to serverless technologies comes with a convenience-over-control trade off. Thus, it is imperative for the developer side to ensure that delegating control will not impede changing business requirements.

Major cloud providers support a wide variety of out-of-the-box serverless services that come with distinct advantages and limitations. One of the earliest FaaS was AWS Lambda, which was launched by Amazon in 2014, and then followed by Google Cloud Functions, Microsoft Azure Functions, and IBM OpenWhisk launched in 2016 [14]. Apart from proprietary products there are several open source projects, such as OpenLambda, OpenFaas, OpenWhisk, Knative, and Kubeless [16–19]. Even though open source alternatives afford higher independence from the proprietary solutions, they are primarily self-hosted, which, in effect, carries the burden of managing the infrastructure.

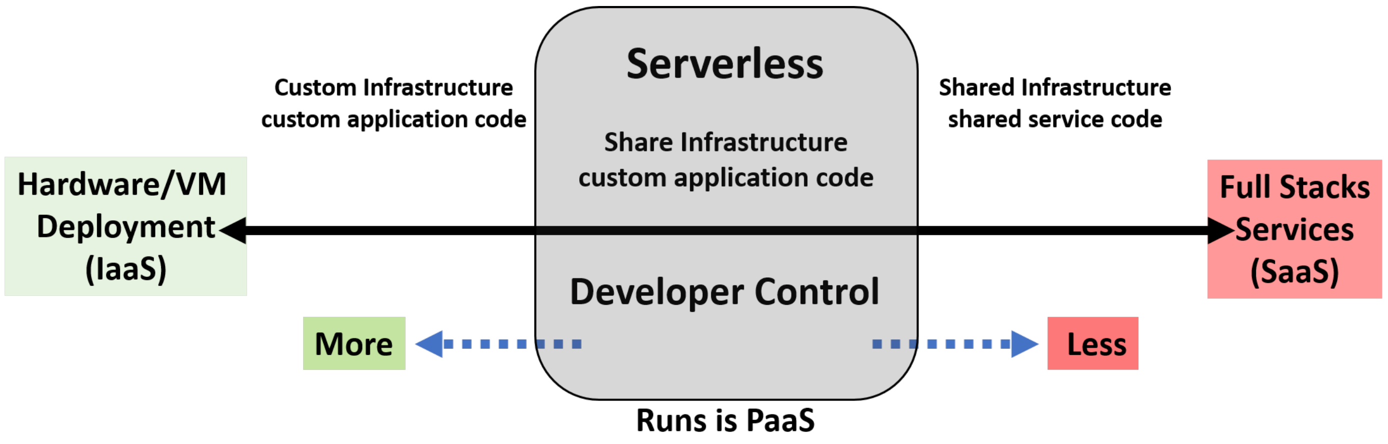


Figure 1. Developer control and serverless computing (adapted from [11]).

Table 1. Comparison of Traditional vs Serverless ETL solutions.

	Traditional ETL	**Serverless ETL**
Maintenance	Traditional ETL pipelines require regular infrastructure maintenance to ensure the Quality of Service. This is usually managed by another dedicated team.	Infrastructure maintenance and updates is taken care by the serverless components' vendor.
On-Demand	Require provisioning and the pricing is the same whether the resources are utilized or not.	Serverless ETL pipelines can be used/created On-Demand which lead to a flexible and dynamic pay per data processed model pricing model.
Cost Effectiveness	Infrastructural components provisioned for traditional ETL solutions are usually priced at a very high level of utilisation.	On the contrary, Serverless cloud services usually have a very granular pricing which results into high cost effectiveness in the long term. i.e., AWS Lambda functions are now billed per 1 ms of execution.
Data Sources	Data sources are usually constrained to a limited set of integrations of structured data.	Can be adapted to almost any data source/format as long as there is a clearly defined approach of transforming it.
Processing	Mainly batch processing of records which leads to only near real-time applications.	Allow for stream processing which can accommodate real-time applications
Pipeline Configuration	Have a very limited set of options to configure the ETL pipeline and it heavily depended on and defined by the implementation. Usually only defined in a single programming language.	Serverless ETL pipelines configuration is still heavily depended on the implementation but there is significantly less technical effort on enabling polyglot implementations.
Elasticity	Scaling horizontally requires provisioning beforehand and careful planning to ensure the smooth integration of new resources.	Serverless components are very elastic allowing for almost instant horizontal and vertical scaling

3. Related Work

Already, multiple studies have focused on harnessing the power of serverless technologies in order to modernise traditionally tedious tasks. Many evaluated performance across different dimensions, others addressed scalability issues, and others introduced novel paradigms. Hence, the related literature can be categorised into evaluations and applications of serverless technologies, yet there is often an overlap between the two.

A seminal example on the application side is PyWren [20], which introduced a MapReduce primitive that was built on top of AWS lambda functions. PyWren provided a simple, highly parallel, serverless data processing system, which showcased a performance comparable to Spark. However, PyWren is not event-driven, still requires external orchestration, and only addresses the data processing stage. Similarly, ExCamera [21] applies the same principle in order to leverage extreme parallelism for digital video encoding, but, yet again, also requires a centralized orchestrator for functions. Stanford researchers also worked on a serverless orchestration framework, called gg [22]. It is a more generalisable approach that still requires an external coordinator, yet it only targets to improve highly-parallelised tasks rather than event-driven data processing. Flint [23], on the other hand, directly focuses on distributed data processing by implementing a PySpark scheduler on AWS Lambda functions and SQS queues for maintaining state. Although the design is convenient due to Spark interoperability, it does not provide a comprehensive evaluation and, thus, comes with challenges in its practicality. Mijanrur et al. supported the idea of using serverless computing for ETL pipelines and presented a very high level overview of an exemplar Serverless ETL architecture on AWS; however, their work did not include any evaluation metrics [24]. Motivated by low maintenance and cost efficiency, Zang et al. proposed a serverless AWS architecture targeting particularly power grid emergency generation dispatch [25]. Unlike previous studies, Perez et al. [26] proposed an event-driven serverless data processing architecture that was built on top of Kubernetes and OpenFaaS. While this is a novel approach, the effort that is required to be deployed and maintained is substantial; minimising that effort is one of the aspects that our reference architecture aims to directly address. Finally, several authors focus on evaluating serverless computing over major cloud providers by exploring performance, consistency, elasticity, security, latency, and task variation, but they are constrained to a single serverless technology, rather than an architecture, as this work [27–29].

As presented by the existing literature, many scholars actively explore how far serverless technologies can be stretched. Some target insurmountable parallelization and others focus on specific use cases. In this context, we extend the state-of-the-art in this field of research by adding our proposed to architecture to the set of practical serverless systems for general ETL data pipelines.

4. Architecture

We chose the AWS platform for our experimental evaluation for several reasons. Firstly, according to a Canalys report, AWS holds the largest market share amongst the major cloud providers [30]. Furthermore, AWS lambdas cold-start times have been proven to be more consistent across multiple invocations [28] and they provide better CPU performance, network bandwidth, and I/O throughput as implementations scale [29]. Other empirical results suggest that the Amazon S3 offers low latency and comparable throughput over other providers [31]. We selected Python3.7 to run our experiments, given that Python is one of the most popular AWS Lambda runtime choices amongst academic studies [32].

Our event-driven ETL pipeline follows a rather basic architecture. The primary aim of this research is to study the behaviour, overheads, and costs in the context of serverless event-driven pipelines, in a manner that is as platform-agnostic as possible: therefore, adopting a straightforward architecture excludes any optimising features AWS may support. Similar to a typical ETL pipeline, the subject architecture consists of a data source, a processing stage, and a data sink. Two additional intermediate stages were implemented for connecting the processing with the source and the sink, primarily for observability and fault tolerance. For reference, we labelled these stages as 'triggering' and 'writing'. Figure 2 gives a high-level overview of the whole architecture, which is outlined here:

Data Source: We use an AWS S3 bucket as a data source. Using S3 is both a cost-effective and scalable choice as it provides low latency and out-of-the-box lambda integration capabilities. While, technically, data could be sent directly to lambda functions while using an API Gateway, S3 acts both as a data source and datalake. Hence, the data persist and they can be processed again in the event of system failure, or act as input for other data analysis pipelines.

Triggering: S3 supports notifications that directly trigger a lambda function. Notably, this is less desirable in live production environments, as it delegates the burden of implementing a fail-over strategy down to the function. Yet, our implementation uses SNS topics and SQS queues in order to invoke the processing stage. Although this increases architectural and network overhead, it is an essential element of the architecture in order to achieve observability and fault tolerance. More precisely, SQS triggers a lambda function synchronously and, hence, any inputs that resulted in a failed invocation will be forwarded to a dead letter queue. What is more, this architectural design allows for a fanout message forwarding, as opposed to normal S3 notifications.

Data Processing: we employ stateless lambda functions for data processing. Ensuring that the processing is stateless poses some limitations on the type of operations that can be applied on the data, due to the lack of global context. There are ways to achieve statefulness amongst many lambda functions [33,34], but these are not within the scope of this work, as they introduce architectural complexities.

Data Writing: the same as in the triggering stage, data writing provides an extra layer of fault tolerance and failover resolution by using an SQS and a lambda function, which is responsible for persisting the data into DynamoDB. It must be noted that SQS limits message sizes to a maximum of 256 KBs by design; therefore, each lambda transformation worker must chunk the transformation output prior to sending it to the queue. This design permits extremely high parallelisation, as each chunk is written by a separate writer. Furthermore, it opens up the opportunity for applying some common processing logic to all of the data after they have been processed (i.e., validation).

Data Sink: we use DynamoDB to store the processed data. DynamoDB is a serverless NoSQL database and it fulfils all our experimental requirements. It is serverless, offers a pay-on-demand plan, it has low latency, provides de-duplication and fast querying capabilities, and it can be very easily extended with further data processing pipelines through Dynamo Streams.

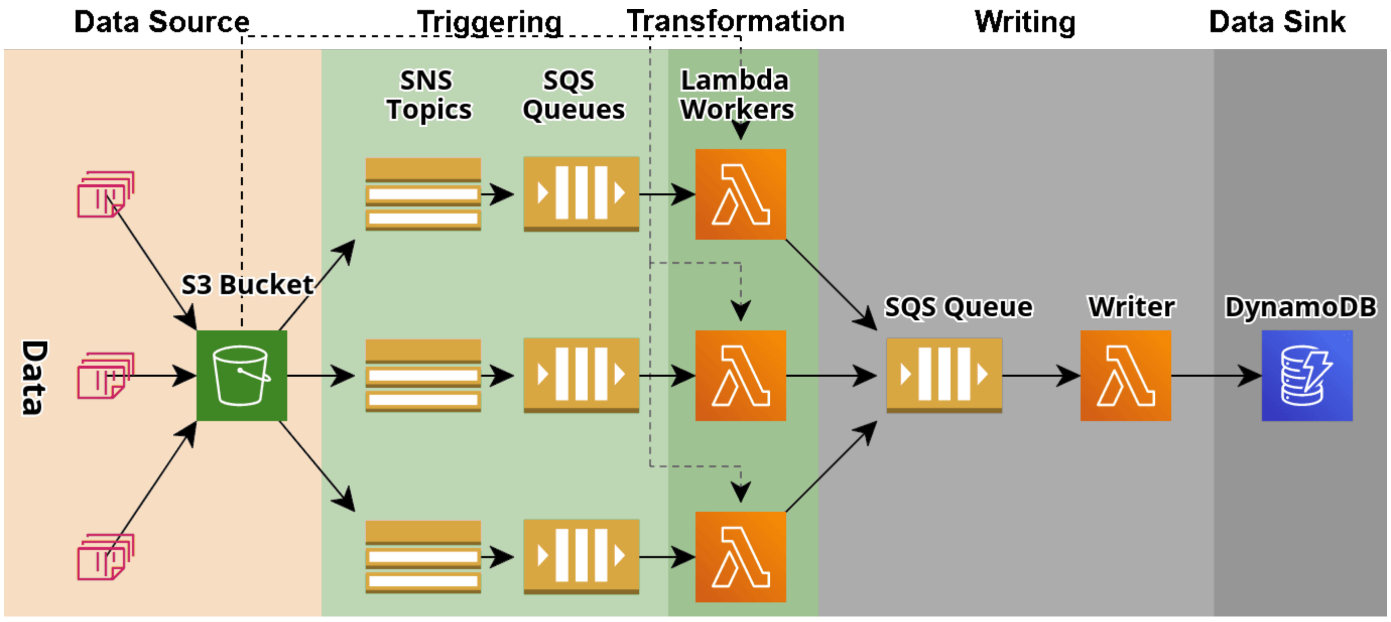


Figure 2. Overview of system architecture.

Pricing

In principle, serverless architectures should feature negligible costs when the system is idle. An ideal alternative would fully eliminate costs, but this is not possible with the current AWS platform capabilities. To have a more granular reference to the costs, we define chargeable elements as anything that is charged as part of AWS services. That being said, we categorise the chargeable elements of the architecture as active or passive, based on how it is charged. Active costing refers to elements that are charged according to data flow usage, while passive costing refers to charges that are applied, even when the system is idle. An example of a passive chargeable element is S3 storage. Another, not very obvious, passive chargeable element is SQS polling requests: when lambda functions are being triggered by SQS, they keep polling the SQS queue, even when there are not any available messages. The standard rate of polling is 15 requests per minute by design. Table 2 lists active and passive chargeable elements across the proposed architecture.

Table 2. Active and passive chargeable elements of architecture.

	Data Source	**Triggering**		**Transformation**	**Data Writing**		**Sink**
	S3	**SNS**	**SQS**	**Lambda**	**SQS**	**Lambda**	**DynamoDB**
Active	Requests (S3Req) Data In (S3In) Data Out (S3Out)	Messages (SNSmsg)	Requests (SQSReq)	GB/Seconds (GBSecCost)	Requests (SQSReq)	GB/Seconds (GBSecCost)	Write Units (DBWrites)
Passive	Storage	-	Requests	-	Requests	-	Storage

5. Experimental Design

We design our experiments to investigate several aspects of the system across different data volumes, event frequencies, and processing power. These are critical factors that should be considered when migrating to serverless configurations. Our empirical evaluation pinpoints the I/O and processing overheads at high granularity. In particular, by investigating how the architecture performs with respect to the following elements, we attempt to answer the corresponding questions:

Consistency: Are pipeline executions consistent?

Reliability: What is the probability that the system delivers the data in its entirety? How often are events lost (if any)?

Pricing: How chargeable elements scale across pipeline runs?

Performance: What is the end to end time of data flow? How do lambda function execution times change across different data volumes and velocities?

Payload size: What is the maximum payload that a pipeline can handle?

For the initial phase of the experiments, we control the total data size to be sent, the payload size for each event, the available memory for the AWS lambda worker, and the frequency of the events. The event payload size is calculated, such that the sum the payloads equals the total size of data to be sent within an hour of running, based on the given frequency. More formally: let f, T, p, and D be, respectively: the frequency, the time that the experiment is running for, payload size per event, and total data size to be sent, then:

$$p = \frac{D}{f \times T} \quad \text{where} \quad T = 60 \ \text{min}$$

We record the relevant metrics for each experiment question across 60 distinct pipeline runs that arise from all possible combinations of the following dependent variables:

D**:** Total data size to be sent (MB): 1, 10, 100.

f**:** Event frequency (events per minute): 1, 2, 6, 30, 60.

M**:** Lambda worker available memory (MB): 128, 256, 512, 1024.

A secondary series of experiments was performed, where a single event was sent to the system in order to address event payload size and performance. During this process the end-to-end time from source to sink was recorded. Specifically, a total of 32 runs were performed in order to cover the following dependent variables:

p**:** Event Payload Size (MB) : 1, 5, 10, 20, 40, 60, 80, 100.

M**:** Lambda worker available memory (MB): 128, 256, 512, 1024.

Because the payload and processing task of the writer function does not vary, the available memory for the writer lambda function is set to 128 MB across all of the experiments. Similarly, the data processing task that the lambda worker carries out remains the same for all experiments. To keep it consistent and quantifiable, we employed a simple pass-through of the data without any extra I/O overhead, which translates to O(N) complexity. Therefore, we can argue that any similar O(N) task can be subjected to our results. All of the Python implementations for the tasks that were carried out by Lambda functions were developed while using Pandas python package v1.1.0 [35] and Python standard libraries.

6. Discussion

6.1. Reliability

The official AWS documentation states that S3 events will most likely deliver an event once, but it is possible that an event will be delivered multiple times, or not at all. The documentation does not clearly quantify how often this may occur and, in order to cater for this, we recorded any such occurrences through the experiments. Judging by our results, S3 delivered the event notification successfully 100% across all experiments (71,280 events). Nonetheless, we noticed two unexplained failed invocations when transferring a total of 100 MB at 1 event per minute using 128 MB of lambda memory (See Figure 3). The functions logs indicated that those functions timed-out when attempting to send the output data chunks to writer SQS queue. Although it is not clear why this time-out occurred, it is likely that this could have been a minor outage of that particular SQS queue, which leads to the conclusion that it is always necessary to establish failover strategies for production environments. With our proposed architecture, this was picked up by the triggering dead letter queue.

6.2. Consistency

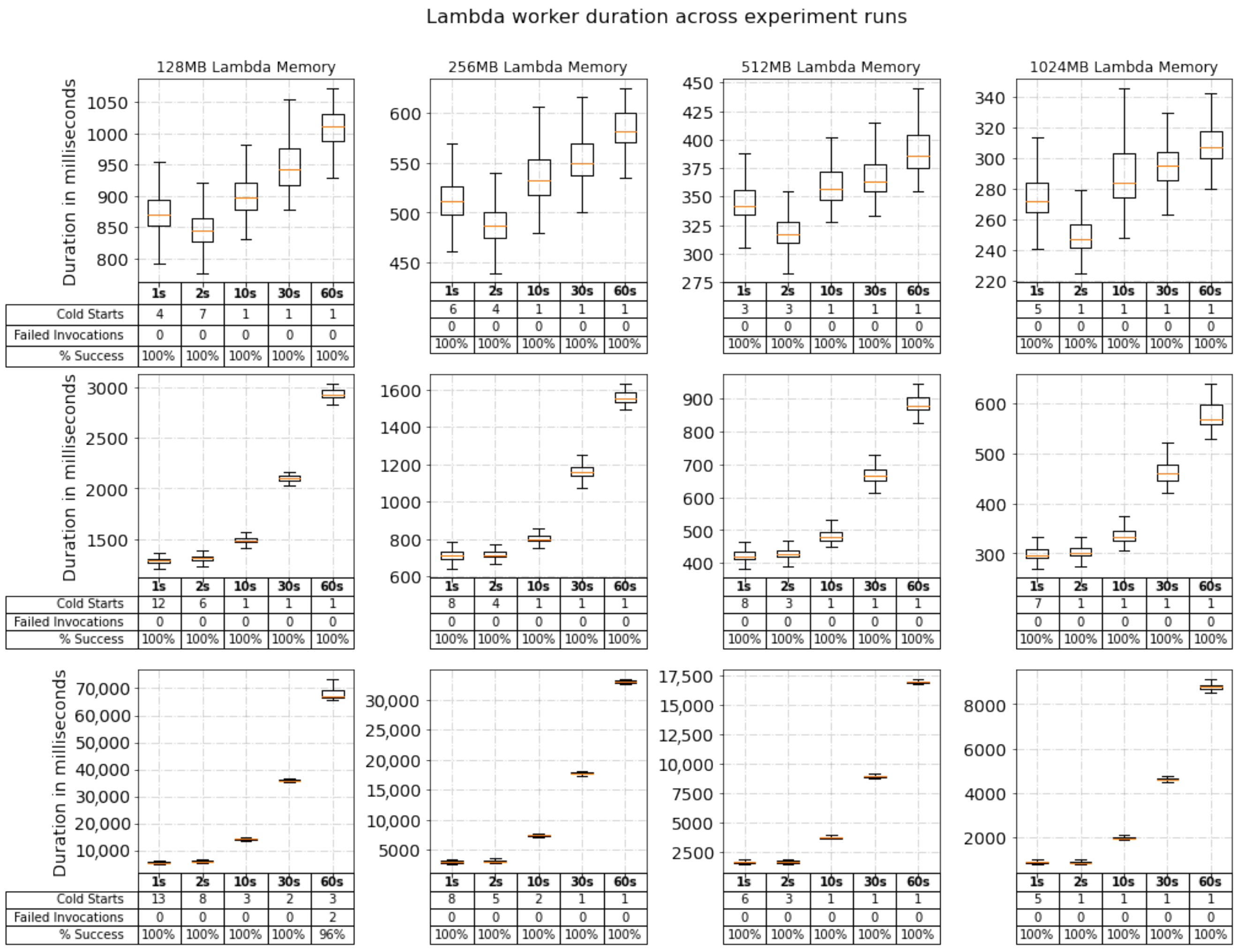
It is imperative that each workflow in an ETL pipeline executes in a consistent manner. Provided that the execution environment is not within our control, the consistency of the system in terms of execution duration per function task and latencies was measured.

The high-level overview of invocation duration times in Figure 3 supports the lambda workers' pipeline execution environment is consistent. As a matter of fact, the average standard deviation through all successful invocations did not exceed 160 ms.

6.3. Performance

We monitored the execution time on each subtask that was implemented in the worker lambda function, bearing in mind that AWS does not provide clear specifications, apart from memory, in their execution environments. Generally, as expected, higher memory functions perform orders of magnitude faster in terms of raw processing. More precisely, the results suggest that the processing power is linearly proportional to the memory since the execution time halves with each memory upgrade.

Figure 4 illustrates the end-to-end times from the source to sink of a single event increasing in size. Specifically, for an event with a payload size of 100 MB, it takes slightly more than 2 min. to process and persist in data sink, which results in around 750 KB/s processing speed that is acceptable, but could be problematic at high data volumes. The cause of this overhead is discussed further below.



128MB Lambda Memory	1s	2s	10s	30s	60s
Cold Starts	4	7	1	1	1
Failed Invocations	0	0	0	0	0
% Success	100%	100%	100%	100%	100%

256MB Lambda Memory	1s	2s	10s	30s	60s
Cold Starts	6	4	1	1	1
Failed Invocations	0	0	0	0	0
% Success	100%	100%	100%	100%	100%

512MB Lambda Memory	1s	2s	10s	30s	60s
Cold Starts	3	3	1	1	1
Failed Invocations	0	0	0	0	0
% Success	100%	100%	100%	100%	100%

1024MB Lambda Memory	1s	2s	10s	30s	60s
Cold Starts	5	1	1	1	1
Failed Invocations	0	0	0	0	0
% Success	100%	100%	100%	100%	100%

128MB	1s	2s	10s	30s	60s
Cold Starts	12	6	1	1	1
Failed Invocations	0	0	0	0	0
% Success	100%	100%	100%	100%	100%

256MB	1s	2s	10s	30s	60s
Cold Starts	8	4	1	1	1
Failed Invocations	0	0	0	0	0
% Success	100%	100%	100%	100%	100%

512MB	1s	2s	10s	30s	60s
Cold Starts	8	3	1	1	1
Failed Invocations	0	0	0	0	0
% Success	100%	100%	100%	100%	100%

1024MB	1s	2s	10s	30s	60s
Cold Starts	7	1	1	1	1
Failed Invocations	0	0	0	0	0
% Success	100%	100%	100%	100%	100%

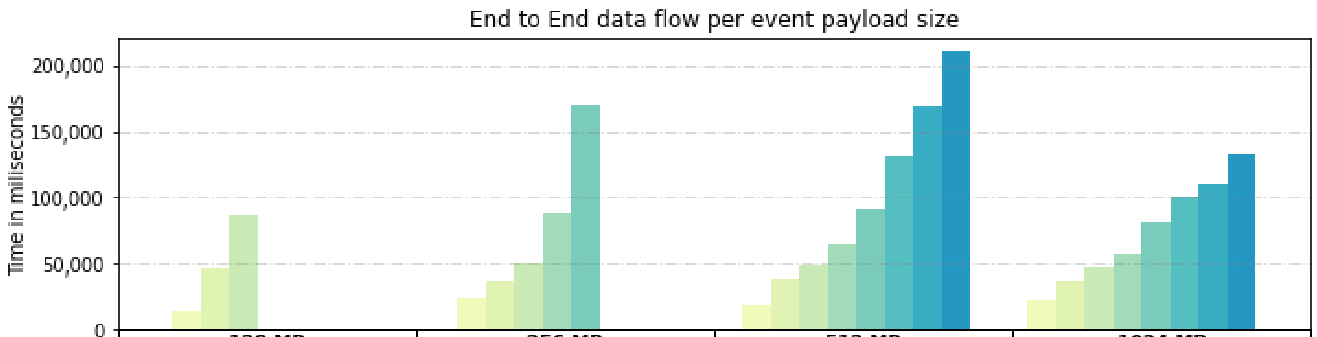
128MB	1s	2s	10s	30s	60s
Cold Starts	13	8	3	2	3
Failed Invocations	0	0	0	0	2
% Success	100%	100%	100%	100%	96%

256MB	1s	2s	10s	30s	60s
Cold Starts	8	5	2	1	1
Failed Invocations	0	0	0	0	0
% Success	100%	100%	100%	100%	100%

512MB	1s	2s	10s	30s	60s
Cold Starts	6	3	1	1	1
Failed Invocations	0	0	0	0	0
% Success	100%	100%	100%	100%	100%

1024MB	1s	2s	10s	30s	60s
Cold Starts	5	1	1	1	1
Failed Invocations	0	0	0	0	0
% Success	100%	100%	100%	100%	100%

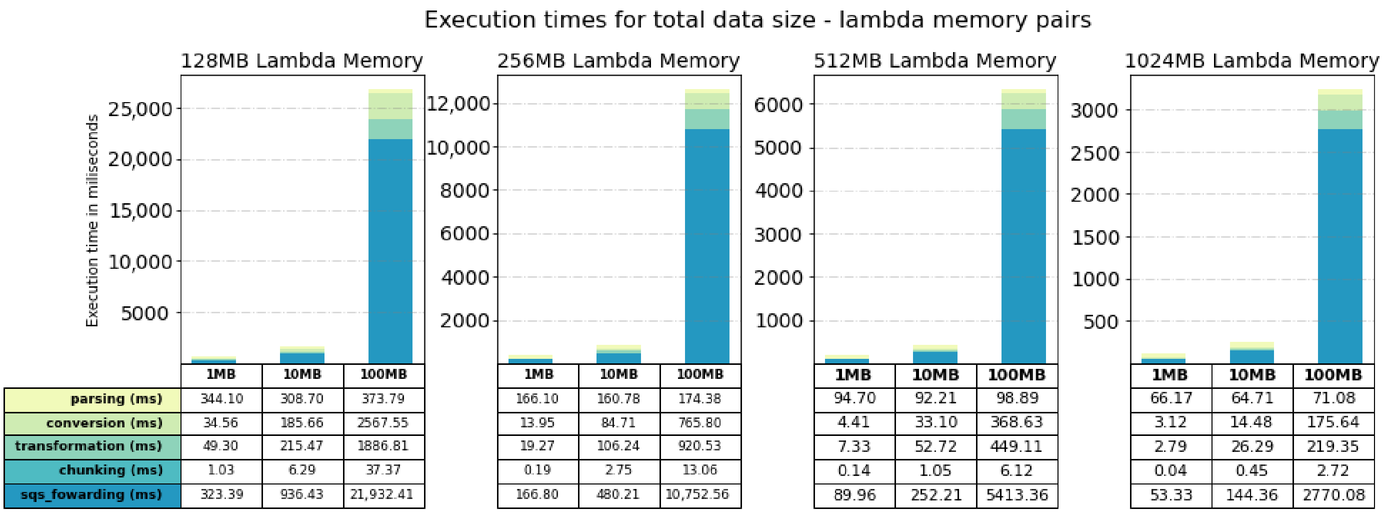
Figure 3. Mean execution times, failed invocations, cold starts, and success rate for distinct Lambda worker operation categorised by lambda function's available memory.



	128 MB	256 MB	512 MB	1024 MB
1 MB	14.462	24.306	17.473	21.662
5 MB	45.387	36.607	36.928	35.807
10 MB	85.879	50.210	48.800	47.199
20 MB	-	88.572	64.126	57.045
40 MB	-	169.731	90.307	80.902
60 MB	-	-	130.978	99.808
80 MB	-	-	169.315	110.100
100 MB	-	-	210.814	132.859

Figure 4. End to end data flow times for event payload size and lambda worker memory. Table rows show time in seconds per payload size. Cells without a value indicate that the lambda worker failed to process the payload.

Similarly, Figure 5 demonstrates the time that is spent performing on each worker subtask. In larger data volumes, sending the data to SQS takes up most of the invocation time, as it consists of blocking network operations. This is due to the need for chunking the data in order to overcome the SQS payload size limits. Alternatives are discussed in future work, as that intensive network operation can undermine successful processing of larger data volumes. Notably, while the worker function forwards the transformed data to the queue, the writer function is invoked in parallel upon notification of the first chunk. Consequently, the total end-to-end data delivery time is reduced.



	128MB Lambda Memory			256MB Lambda Memory			512MB Lambda Memory			1024MB Lambda Memory		
	1MB	10MB	100MB	1MB	10MB	100MB	1MB	10MB	100MB	1MB	10MB	100MB
parsing (ms)	344.10	308.70	373.79	166.10	160.78	174.38	94.70	92.21	98.89	66.17	64.71	71.08
conversion (ms)	34.56	185.66	2567.55	13.95	84.71	765.80	4.41	33.10	368.63	3.12	14.48	175.64
transformation (ms)	49.30	215.47	1886.81	19.27	106.24	920.53	7.33	52.72	449.11	2.79	26.29	219.35
chunking (ms)	1.03	6.29	37.37	0.19	2.75	13.06	0.14	1.05	6.12	0.04	0.45	2.72
sqs_fowarding (ms)	323.39	936.43	21,932.41	166.80	480.21	10,752.56	89.96	252.21	5413.36	53.33	144.36	2770.08

Figure 5. Mean execution times for distinct lambda worker operation separated by Lambda function's available memory.

6.4. Payload Size

Another important element of the architecture are the limitations that are imposed on the payload size of a single event. Worker functions with 128 MB and 256 MB of memory failed to process payload sizes larger than 10 MB and 40 MB, respectively, due to lack of processing power, as illustrated in Figure 4. On the contrary, 512 MB and 1024 MB memory workers are shown to be able to handle payloads of sizes up to 100 MB and potentially even larger.

6.5. Pricing

In any serverless system, cost-awareness is critical. Although calculating the pricing for a single resource is straightforward, calculating the costs for the whole system can be a daunting task. For that purpose we present a simple expression that models the costs of the aforementioned architecture. Let $D = \sum_{i=0}^{n} p_i$ where p_i is the ith payload size of n events received over time t, (S3Req, SNSmsg, SQSreq, S3In, S3Out, GBs, DBWrites) be the pricing of each chargeable element accordingly, and Wcost, WRcost, Wout, and WRrecords be, respectively, the worker's/writer's duration in GB/Seconds, size of transformed output, and number of records to be written in the DB, as also provided in Table 2. Subsequently, the active costs can be summarised as:

$$\begin{aligned} Cost = n \times (S3Req + SNSmsg + SQSreq) + D \times (S3In + S3Out) + \\ (Wcost + WRcost) \times GBSecCost + (\frac{Wout}{256\ \text{KB}} + 1) \times SQSreq + WRrecords \times DBWrites \end{aligned}$$

More specifically, the division over 256 KB chunks models the fragmentation of SQS messages to match the maximum message payload size of AWS (256 KB). Note that this is not a complete model, as it neglects passive costs and redrive policies in the case of errors. However, it affords us the observation that many of the costs have linear relationships with the pipeline inputs. Still, the lambda function charges (*Wcost*, *WRcost*) are somewhat harder to estimate, since they are affected by several factors. With this in mind, in Figure 6, we empirically present the pricing attributed to the worker lambda function throughout the experiments. Interestingly, it is observed that, in higher data volumes, tuning of the lambda functions specifications is essential, since there are cases where a more powerful lambda function costs less than an underspecified function (e.g., 128 MB/256 MB Memory at one event per minute).

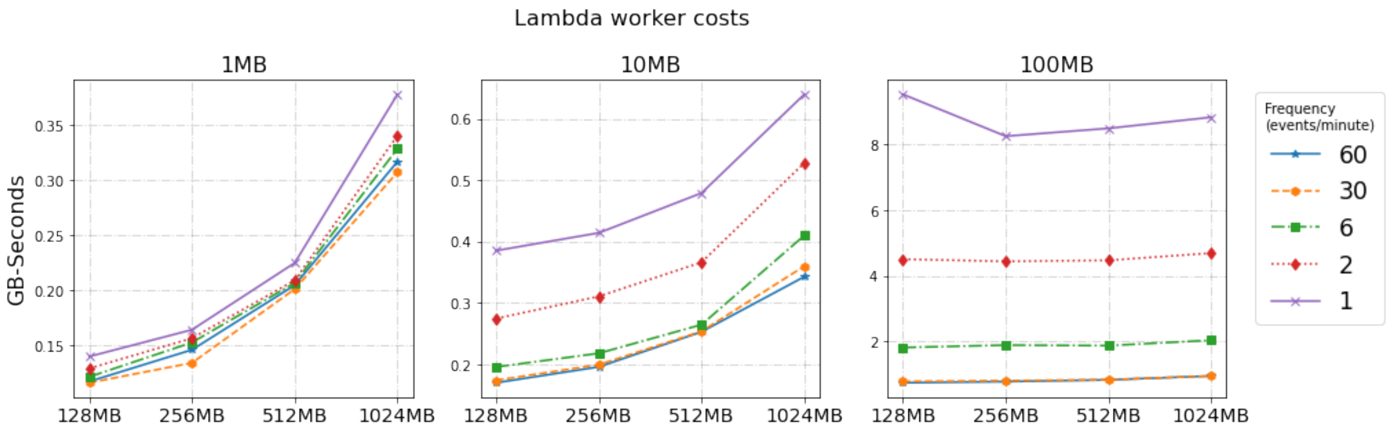


Figure 6. Lambda worker costs in GB/Seconds.

7. Limitations and Future Work

The discussion of this work's results highlights the capabilities of the proposed system across many aspects. In particular, we demonstrated that the proposed architecture can cope reliably with large event payloads for frequencies up to one event per second. What is more, we highlight the cost effectiveness of this serverless implementation by modeling the chargeable components of the architecture and underlining their dynamic pay-as-you- go nature, as opposed to fixed pay rates of serverful architectures. Nevertheless, we summarise the limitations of our architecture and expand on future work to address these. The main deficiency of our architecture is the SQS data throughput bottleneck. This is naturally inherited by the AWS SQS 256 KB message size restriction. Furthermore, in our methodology, we intentionally omit the pricing of components that are used for pipeline failover, such as dead letter queues. Finally, the lack of lambda worker orchestration requires that each ETL pipeline must be guaranteed in order to terminate before the execution time of the AWS lambda functions runs out.

This research can be extended in a variety of ways. Regarding the evaluation and architecture design, there is already a list of parameters in our backlog that we pursue to tune and experiment. One of the top future work prioritiez is to overcome the limitations of the proposed architecture by implementing alternatives to SQS data transfer. A possible workaround would be to keep using SQS for triggering, but routing the data through S3 or adopting an asynchronous approach for delivering SQS messages. Both of these approaches would have their pros and cons and, hence, they are at the top of the future tasks, for a more systematic investigation. In addition, in order to overcome the intrinsic limitation of execution time that is imposed by the AWS Lambda function service, we are planning to extend the proposed architecture, such that it can automatically orchestrate long running ETLs to multiple interconnected Lambda function invocations. Adding to this, improvements may be sought in the writing stage, and in terms of how the data are chunked. Adding compression or further function orchestration may potentially eliminate some of the limitations discussed in Section 6.4. Although this research provides evidence of resilience against uniform event payloads, further investigation on the limits of the architecture as frequency grows unpredictably would provide useful insights. Finally, we plan to expand our evaluation with similar architectures that are built on other major cloud platforms, such as Azure, GCP, and IBM, as comparison may allow for further conclusions to be drawn.

8. Conclusions

In this study, we developed and presented a reference architecture for building an event-driven ETL pipeline on top of AWS while using entirely serverless technologies, hence enabling a pay-per-usage model. This was accommodated by an empirical evaluation, which concluded that the subject architecture provides consistency, reliability, and acceptable performance for practical use in applications up to one event per second. The study finally recommended a fundamental pricing model in order to approximate the cost of chargeable active tasks in the pipeline.

The applicability of this work extends to fields, such as healthcare, fintech, traffic control, IoT sensor analysis, and many more. Generally, the presented architecture is a good fit for event-based reactive systems that seek to employ flexible ETL pipelines, without the accompanying infrastructural overhead. Our model would provide an extremely cost effective and practical solution for sparse event processing , such as patient journey mapping [36], credit card fraud detection [37], or smart city sensor data analysis [38].

Author Contributions: This paper was accomplished based on collaborative work of the authors. A.P. performed the experiments and analyzed the data. Experiment interpretation and paper authorship were jointly performedby A.P and G.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available on request from the corresponding authors. The experiments use data generated for simulation purposes.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AWS	Amazon Web Services
ETL	Extract-Transform-Load
FaaS	Function-as-a-Service
GCP	Google Cloud Platform
QoS	Quality of Service
SQS	Simple Queue Service
SNS	Simple Notification Service
S3	Simple Storage Service

References

1. Kreps, J.; Corp, L.; Narkhede, N.; Rao, J.; Corp, L. Kafka: A distributed messaging system for log processing. *Proc. NetDB* **2011**, *11*, 1–7.
2. Apache Flink: Stateful Computations over Data Streams. 2011. Available online: https://flink.apache.org/ (accessed on 24 September 2020).
3. Apache Flume. 2011. Available online: https://flume.apache.org/ (accessed on 24 September 2020).
4. Apache Airflow. 2015. Available online: https://airflow.apache.org/ (accessed on 26 September 2020).
5. Naeem, M.A.; Dobbie, G.; Webber, G. An event-based near real-time data integration architecture. In Proceedings of the 2008 12th Enterprise Distributed Object Computing Conference Workshops, Munich, Germany, 16 September 2008; pp. 401–404.
6. Taibi, D.; El Ioini, N.; Pahl, C.; Niederkofler, J.R.S. Patterns for Serverless Functions (Function-as-a-Service): A Multivocal Literature Review. In Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER 2020), Prague, Czech Republic, 7–9 May 2020; pp. 181–192.
7. Hong, S.; Srivastava, A.; Shambrook, W.; Dumitraș, T. Go serverless: Securing cloud via serverless design patterns. In Proceedings of the 10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18), Boston, MA, USA, 9 July 2018.
8. Stafford, G. Event-Driven, Serverless Architectures with AWS Lambda, SQS, DynamoDB, and API Gateway | Programmatic Ponderings. 2019. Available online: https://programmaticponderings.com/2019/10/04/event-driven-serverless-architectures-with-aws-lambda-sqs-dynamodb-and-api-gateway/ (accessed on 15 September 2020).

9. Enriching Event-Driven Architectures with AWS Event Fork Pipelines | AWS Compute Blog. 2019. Available online: https://aws.amazon.com/blogs/compute/enriching-event-driven-architectures-with-aws-event-fork-pipelines/ (accessed on 27 September 2020).
10. Kulmi, M.K. Building Serverless ETL Pipelines on AWS. 2020. Available online: https://www.impetus.com/blog/cloud/building-serverless-etl-pipelines-aws (accessed on 23 September 2020).
11. Baldini, I.; Castro, P.; Chang, K.; Cheng, P.; Fink, S.; Ishakian, V.; Mitchell, N.; Muthusamy, V.; Rabbah, R.; Slominski, A.; et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*; Springer: Berlin/Heidelberg, Germany, 2017; pp. 1–20.
12. 2016 Future of Application Development and Delivery Survey. 2016. Available online: https://www.nginx.com/resources/library/app-dev-survey/ (accessed on 15 September 2020).
13. Aljabre, A. Cloud computing for increased business value. *Int. J. Bus. Soc. Sci.* **2012**, *3*, 234–239.
14. Castro, P.; Ishakian, V.; Muthusamy, V.; Slominski, A. The server is dead, long live the server: Rise of Serverless Computing, Overview of Current State and Future Trends in Research and Industry. *arXiv* **2019**, arXiv:1906.02888.
15. Hellerstein, J.M.; Faleiro, J.; Gonzalez, J.E.; Schleier-Smith, J.; Sreekanti, V.; Tumanov, A.; Wu, C. Serverless computing: One step forward, two steps back. *arXiv* **2018**, arXiv:1812.03651.
16. Hendrickson, S.; Sturdevant, S.; Harter, T.; Venkataramani, V.; Arpaci-Dusseau, A.C.; Arpaci-Dusseau, R.H. Serverless Computation with OpenLambda. In Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16), Denver, CO, USA, 22–24 June 2016; USENIX Association: Denver, CO, USA, 2016.
17. OpenFaaS: Serverless Functions Made Simple. 2019. Available online: https://www.openfaas.com/ (accessed on 21 September 2020).
18. Knative. 2018. Available online: https://knative.dev/ (accessed on 21 September 2020).
19. Kubeless. 2018. Available online: https://kubeless.io/ (accessed on 30 September 2020).
20. Jonas, E.; Pu, Q.; Venkataraman, S.; Stoica, I.; Recht, B. Occupy the cloud: Distributed computing for the 99%. In Proceedings of the 2017 Symposium on Cloud Computing, Santa Clara, CA, USA, 24–27 September 2017; pp. 445–451.
21. Fouladi, S.; Wahby, R.S.; Shacklett, B.; Balasubramaniam, K.V.; Zeng, W.; Bhalerao, R.; Sivaraman, A.; Porter, G.; Winstein, K. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), Boston, MA, USA, 27–29 March 2017; pp. 363–376.
22. Fouladi, S.; Romero, F.; Iter, D.; Li, Q.; Chatterjee, S.; Kozyrakis, C.; Zaharia, M.; Winstein, K. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In Proceedings of the 2019 USENIX Annual Technical Conference (USENIXATC 19), Renton, WA, USA, 10–12 July 2019; pp. 475–488.
23. Kim, Y.; Lin, J. Serverless Data Analytics with Flint. In Proceedings of the 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), San Francisco, CA, USA, 2–7 July 2018; pp. 451–455.
24. Rahman, M.M.; Hasan, M.H. Serverless Architecture for Big Data Analytics. In Proceedings of the 2019 Global Conference for Advancement in Technology (GCAT), Bangalore, India, 18–20 October 2019; pp. 1–5.
25. Zhang, S.; Luo, X.; Litvinov, E. Serverless computing for cloud-based power grid emergency generation dispatch. *Int. J. Electr. Power Energy Syst.* **2021**, *124*, 106366. [CrossRef]
26. Pérez, A.; Risco, S.; Naranjo, D.M.; Caballer, M.; Moltó, G. On-Premises Serverless Computing for Event-Driven Data Processing Applications. In Proceedings of the 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), Milan, Italy, 8–13 July 2019; pp. 414–421.
27. Kuhlenkamp, J.; Werner, S.; Borges, M.C.; El Tal, K.; Tai, S. An Evaluation of FaaS Platforms as a Foundation for Serverless Big Data Processing. In Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing, UCC'19, Auckland, New Zealand, 2–5 December 2019; Association for Computing Machinery: New York, NY, USA, 2019; pp. 1–9. [CrossRef]
28. Wang, L.; Li, M.; Zhang, Y.; Ristenpart, T.; Swift, M. Peeking behind the curtains of serverless platforms. In Proceedings of the 2018 USENIX Annual Technical Conference (USENIXATC 18), Boston, MA, USA, 11–13 July 2018; pp. 133–146.
29. Lee, H.; Satyam, K.; Fox, G. Evaluation of production serverless computing environments. In Proceedings of the 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), San Francisco, CA, USA, 2–7 July 2018; pp. 442–450.
30. Global Cloud Services Market Q2 2020. 2020. Available online: https://canalys.com/newsroom/worldwide-cloud-infrastructure-services-Q2-2020 (accessed on 20 October 2020).
31. Bjornson, Z. AWS S3 vs. Google Cloud vs Azure:Cloud Storage Performance. 2015. Available online: http://blog.zachbjornson.com/2015/12/29/cloud-storage-performance.html (accessed on 10 October 2020).
32. Scheuner, J.; Leitner, P. Function-as-a-Service performance evaluation: A multivocal literature review. *J. Syst. Softw.* **2020**, *170*, 110708. [CrossRef]
33. Barcelona-Pons, D.; Sánchez-Artigas, M.; París, G.; Sutra, P.; García-López, P. On the faas track: Building stateful distributed applications with serverless architectures. In Proceedings of the 20th International Middleware Conference, Davis, CA, USA, 9–13 December 2019; pp. 41–54.
34. Sreekanti, V.; Lin, C.W.X.C.; Faleiro, J.M.; Gonzalez, J.E.; Hellerstein, J.M.; Tumanov, A. Cloudburst: Stateful functions-as-a-service. *arXiv* **2020**, arXiv:2001.04592.

35. Pandas Development Team. Pandas-Dev/pandas: Pandas. 2020. Available online: https://zenodo.org/record/4311557#.X-LqptgzbIU (acessed on 30 September 2020).
36. Arias, M.; Rojas, E.; Aguirre, S.; Cornejo, F.; Munoz-Gama, J.; Sepúlveda, M.; Capurro, D. Mapping the Patient's Journey in Healthcare through Process Mining. *Int. J. Environ. Res. Public Health* **2020**, *17*, 6586. [CrossRef] [PubMed]
37. Arya, M.; G, H.S. DEAL–'Deep Ensemble ALgorithm' Framework for Credit Card Fraud Detection in Real-Time Data Stream with Google TensorFlow. *Smart Sci.* **2020**, *8*, 71–83. [CrossRef]
38. Garcia Alvarez, M.; Morales, J.; Kraak, M.J. Integration and Exploitation of Sensor Data in Smart Cities through Event-Driven Applications. *Sensors* **2019**, *19*, 1372. [CrossRef] [PubMed]