An Empirical Evaluation of Serverless Cloud Infrastructure for Large-Scale Data Processing

Thomas Bodner
Hasso Plattner Institute,
University of Potsdam
Potsdam, Germany
thomas.bodner@hpi.de

Theo Radig
Hasso Plattner Institute,
University of Potsdam
Potsdam, Germany
theo.radig@hpi.de

David Justen*
BIFOLD, TU Berlin
Berlin, Germany
david.justen@tu-berlin.de

Daniel Ritter*
SAP
Walldorf, Germany
daniel.ritter@sap.com

Tilmann Rabl
Hasso Plattner Institute,
University of Potsdam
Potsdam, Germany
tilmann.rabl@hpi.de

ABSTRACT

Data processing systems are increasingly deployed in the cloud. While monolithic systems run fully on virtual servers, recent systems embrace cloud infrastructure and utilize the disaggregation of compute and storage to scale them independently. The introduction of serverless compute services, such as AWS Lambda, enables finer-grained and elastic scalability within these systems. Prior work shows the viability of serverless infrastructure for scalable data processing, but sees limitations due to performance variance and cost overhead, especially in networking and storage.

In this paper, we perform a detailed analysis of the performance and cost characteristics of serverless infrastructure in the data processing context. We base our analysis on a large series of microbenchmarks across different compute and storage services, as well as end-to-end workloads. To enable our analysis, we propose the Skyrise serverless evaluation platform. For the widely used serverless infrastructure of AWS, our analysis reveals distinct boundaries for performance variability in serverless networks and storage. We also present cost break-even points for serverless compute and storage. These insights provide guidance on when and how serverless infrastructure can be used efficiently for data processing.

1 INTRODUCTION

Serverless infrastructure is an increasingly popular foundation for applications in the cloud [56, 60]. Multiple cloud providers offer compute and storage services that abstract from the provisioning and management of servers [3, 42, 69, 92]. Services such as AWS Lambda [16] and S3 [14] allocate fine-grained resources based on consumption, providing more elastic scalability and operational simplicity than conventional cloud infrastructure. The promise of cost efficiency for sporadic usage has spurred the adoption of serverless infrastructure for infrequent and short-running applications. Common examples are web, mobile, and IoT application backends that require little coordination and state management [29, 62, 105].

*Work partially done at Hasso Plattner Institute.

Although large-scale, data-intensive applications benefit from the elasticity of cloud infrastructure [45, 59] and aim for finer-grained elasticity [86, 112], only few build on serverless resources in practice. We attribute this to the sub-optimal performance and cost for data access and communication, as indicated by prior work [100, 106, 114] and elaborated by this work. The elasticity of serverless architectures is enabled by storage disaggregation, which requires access to persistent and ephemeral state via the network. Today, serverless compute is offered as functions as a service (FaaS) [16, 66]. Serverless functions are small and short-lived compute units restricted to keep no state and communicate with one another only indirectly via shared cloud storage.

Despite the limitations, data analysis systems have been built on serverless resources. The FaaS-based systems PyWren [75, 100], Starling [98], and Lambada [93] show scalable performance and cost efficiency for analytical workloads with low query volumes, i.e., inter-arrival times in minutes. For more frequent workloads, they are not cost-competitive. To make serverless infrastructure a viable foundation for more sustained workloads, we need a better understanding of production serverless systems based on thorough evaluation. While some aspects have been studied well, e.g., CPU performance, FaaS platform overheads, and workload concurrency [58, 103, 118], the factors of performance of storage and networking, and variability for processing large data require more attention.

Existing work [93, 95, 100] lacks a detailed analysis of the network performance of serverless functions and does not consider all serverless storage options (cf. [41]). Most experiments are executed at small scale and it is unclear how performance translates to system components. Performance variance is a well-known issue in cloud environments [102, 111] and intensified in the serverless setting. Prior work only studies facets of serverless performance variability [61, 98, 110]. We need a holistic view at the application level across different locations and extended timeframes. Additionally, we need a better understanding of the economic tradeoffs for using serverless cloud resources for data processing. Serverless resources meet demand quickly and closely, but they come with higher unit costs [13, 30].

In this paper, we analyze the performance and cost factors of serverless infrastructure for large-scale data processing with a focus on previously overlooked characteristics. We introduce *Skyrise*, a framework designed to facilitate experimentation in serverless data processing.[1]

[1]Skyrise is open-source and available at https://github.com/hpides/skyrise.

In summary, we make the following contributions:

(1) We present Skyrise, a framework to analyze the performance and cost of serverless cloud infrastructure for large-scale data processing. Skyrise includes a suite of microbenchmarks and a serverless query engine to run end-to-end workloads.
(2) Using Skyrise, we perform an extensive evaluation of serverless networking and storage to characterize bursting and warming effects in the AWS infrastructure and show their impact on analytical applications. Factoring out these effects, we quantify remaining sources of variability.
(3) We compare the cost of query processing in Skyrise with cloud functions versus VMs and identify break-even points for economic viability of serverless compute and storage.

The rest of the paper is structured as follows. In Section 2, we cover important background on serverless infrastructure. Then, we introduce the Skyrise framework in Section 3. In Section 4, we present our performance evaluation results. Section 5 addresses the economic viability of serverless resources. We discuss our findings in Section 6 and related work in Section 7. We conclude in Section 8.

2 SERVERLESS INFRASTRUCTURE

Serverless infrastructure services, such as AWS Lambda and S3, provide access to large multi-tenant pools of resources. They enable their users to consume these resources quickly, so upfront provisioning is often unnecessary to meet workload performance requirements. They bill the resources at fine granularities to minimize the cost of idle capacity. Users do not need to over-provision resources at excessive cost to prevent performance disruptions from demand spikes in dynamic workloads.

This is possible through the pervasive usage of multi-tenancy. Providers of serverless infrastructure place many user workloads on the same physical resources, e.g., CPUs or drives, to achieve high resource utilization and efficiency. They colocate uncorrelated workloads from separate applications and industries to improve the elasticity and economy of their services. Workload decorrelation enables providers to provision for predictable long-term and cross-tenant average demand. It allows users to employ excess capacity of co-tenants to handle demand spikes [51].

Sharing resources between tenants, however, leads to contention which causes variance in performance. To enhance the robustness of their services, providers use multiple techniques, including admission control, adaptive tenant placement [117], and tenant isolation [52]. While every provider implements these techniques differently, serverless infrastructure services possesses inherent bursting, warming, and variability characteristics. Well-documented examples include:

- **AWS Lambda Function Scaling:** Users can start up to 3,000 function instances in an initial burst, after which Lambda scales tenant slots at a rate of 500 per minute [35].
- **AWS DynamoDB IOPS:** Users get burst throughput from up to 5 minutes of unused capacity. Partitions of tenants that constantly exceed capacity are migrated [33, 63].
- **Google Cloud Storage:** Users need to gradually increase request rates to warm up their buckets for load spikes [65]. They must expect high tail latencies for requests [110].

The rest of this section presents the two types of infrastructure that are most essential for data processing, namely FaaS platforms providing compute capacity and serverless storage.

Table 1: Sizing and pricing of AWS compute services.

Resource	**Lambda** (ARM)	**EC2** (C6g[2])
Memory	Configurable [38]	Configurable[3]
Capacity [GiB]	0.125 – 10	2 – 128
Price [¢/GiB-h][4]	3.84 – 4.80	0.65 – 1.70
Compute	Memory-based[5]	Configurable
Capacity [vCPU]	0.07 – 5.79	1 – 64
Price [¢/vCPU-h]	6.79 – 8.49	1.30 – 3.40
Network	Constant	Compute-based [20]
Bandwidth [Gbps][6]	0.63	0.375 – 25
Price [¢/Gbps-h]	0.48 – 0.60	3.27 – 8.70
Storage	Configurable	Configurable
Capacity [GiB]	0.5 – 10	0 – 3,800
Price [¢/GiB-mo]	8.12	2.33 – 5.41

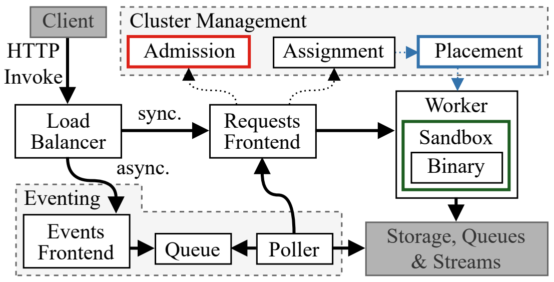

Figure 1: Architecture of the AWS Lambda FaaS platform with control (dotted) and data (solid) paths.

2.1 Function as a Service Platforms

All prominent cloud vendors provide FaaS platforms, such as AWS Lambda, Google Cloud Functions [66], Microsoft Azure Functions [89], and Alibaba Cloud Function Compute [2]. Users of cloud function services upload their application binaries as ZIP archives or container images. They configure function sizes and how functions are invoked.

Current FaaS platforms limit their configuration, as shown in Table 1. Function sizing is usually done based on memory capacity, which determines the number of virtual CPU cores. Compared to VMs, functions are restricted to be about an order of magnitude smaller. FaaS platforms further disallow hour-long lifetime, persistent state, and direct communication. Functions with these characteristics are small, short, and ephemeral tasks that are easy to manage.

Cloud functions are invoked by either users via HTTP requests or triggers on events from queues [28, 76], streams [25, 54], and storage services. The architecture and invocation procedure of the widely used FaaS system Lambda is depicted in Figure 1 [1]. A user request enters the system via a load-balanced frontend service, which coordinates function invocations. The frontend retrieves the function metadata and checks with the admission service (in red) if the invocation exceeds the user's quota for concurrent function executions. Then, the frontend asks the assignment service for a sandboxed environment in the worker fleet to execute the function binary. Sandboxes (in green) are

[2]ARM-based Lambda functions [36] and compute-oriented EC2 C6g instances [10] use Graviton2 processors [115] and have comparable CPU-to-RAM capacity ratios.
[3]The resource capacity ratios and limits depend on the EC2 instance type [12].
[4]We provide ranges for the pricing tiers of Lambda [30] and the prices of EC2 on-demand and reserved instances [13]. All prices are for the AWS us-east-1 region.
[5]A Lambda function gets 1 vCPU equivalent per 1,769 MiB of memory [37, 38].
[6]AWS only provides partial information [10, 38]. We report the baseline bandwidth from Section 4.2. Lambda network bandwidth is constant over instance sizes.

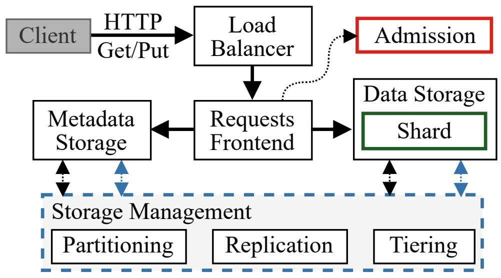

Figure 2: Schematic architecture of a serverless storage system with control (dotted) and data (solid) planes.

implemented with virtualization to run thousands of functions from different users in isolation on each worker machine [1, 37]. If there is a free sandbox, the frontend routes the request payload to this sandbox for function execution. If no sandbox is available, the placement service is asked to create a new environment on a worker with sufficient capacity. This involves downloading and initializing the binary along with its dependencies and is referred to as a coldstart (the blue path). Other than synchronous requests, both asynchronous requests and events are received by the polling service which polls their payloads from internal and external queues, respectively, and invokes functions as a proxy, adding further latency to the invocation path.

FaaS platforms charge based on the number, size, and duration of function invocations [30]. The invocation duration is typically tracked at a millisecond granularity, compared to the second to minute intervals in VM billing [13]. Depending on the resource type, Lambda functions have 2.5–5.9× higher unit prices than EC2 VMs (cf. Table 1).

2.2 Serverless Storage Services

Cloud users have three options for serverless storage: object stores [14, 87], key-value stores [4, 64], and distributed filesystems [6, 88]. All of these services provide elastically scalable storage capacity with high availability and durability guarantees.

Object storage, such as S3 and Azure Blob Storage, is designed to store immutable binary objects of varying sizes and to access these objects with scalable bandwidth. Key-value stores, such as DynamoDB and GCP Firestore, support lower latency key lookups at higher IOPS for kilobyte-sized values. AWS EFS, Azure Files, and other networked filesystems with an NFS or SMB interface provide an abstraction of files and directories.

The architecture of key-value and object storage systems is illustrated in Figure 2 [63]. Users interact with these systems via a simple HTTP Get/Put API. Their requests go through load-balancing, admission (marked red), and request-routing components before a metadata service maps the requested key to a server in the storage fleet. Then, the storage server either receives or sends the data. Data are partitioned and replicated on many storage servers and servers hold shards (in green) of many users. Partitions that outgrow their size or serve excessive load are split and spread evenly across the fleet (marked in blue). We refer to this process as warming and the inverse process as cooling. For instance, S3 partitions typically serve 3.5–5.5K IOPS before they are split [32].

[7]We assume no costs for data access across regions, zones, or virtual networks [5].
[8]We provide ranges for the pricing tiers of S3 [27] and the prices of EFS types [22].
[9]The S3 Express storage class charges requests for transferred data beyond 512 KiB.

Table 2: Pricing of AWS serverless storage services.

Component	**Requests** [¢/M] Read	Write	**Transfers** [¢/GiB][7] Read	Write	**Storage** [¢/GiB-mo][8]
S3 Standard	40	500	0	0	2.1 – 2.3
S3 Express[9]	20	250	0.15	0.8	16
DynamoDB	25	125	0	0	25
EFS	0	0	3	6	16 – 30

The pricing model of serverless storage services is a composite of prices for data storage, requests, and transfers, as shown in Table 2. In AWS, S3 is by an order of magnitude the cheapest option to store data. S3 request cost are independent of the size (from 1 B to 5 TiB), yet they are the highest among the services. Keeping S3 warm for 100K IOPS costs $144 per hour. The pre-warmed S3 Express variant charges requests based on size, resulting in 24–115× higher prices for the throughput-optimal 8–16 MiB range [32]. In DynamoDB, requests are split and charged in kilobyte-scale units. EFS has the highest data transfer fee.

3 SKYRISE EVALUATION FRAMEWORK

In this section, we introduce the Skyrise evaluation framework for experimentation in serverless data processing. Our framework includes a comprehensive suite of microbenchmarks for serverless resources and integrates a serverless query engine to run application-level benchmarks. The framework automates the setup, execution, and result processing for the experiments in our evaluation. Hence, it enables the reproduction of our experimental results. We give an overview of the design and implementation of the framework in Section 3.1. Then, we describe our prototype query engine for the execution of complete queries in Section 3.2.

3.1 Framework Overview

The core components of our framework (cf. Figure 3) are written in C++ and Python. We use C++ for the performance-critical system drivers and measurement functions. The experiment configurations and result plots are in Python. We integrate our codebase with the AWS infrastructure services Lambda and EC2 for compute, as well as S3, DynamoDB, and EFS as storage options. We choose the AWS environment, because Lambda is the most widely used [60] and studied [103] FaaS platform with the fewest performance-related restrictions [38, 67, 90]. The framework is open-source and enables the integration of additional benchmarks and cloud infrastructure.

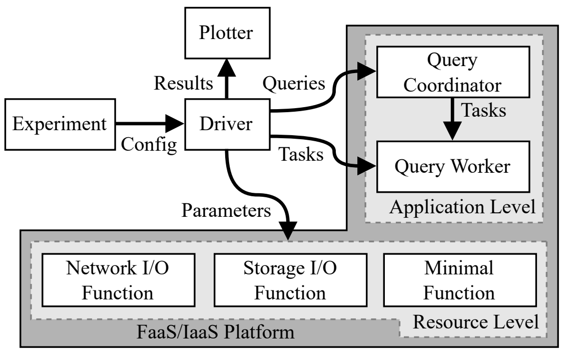

Figure 3: Architecture of the Skyrise evaluation framework showing the experiment execution flow from config to plot.

The framework supports experiments on two levels in the stack. On the resource level, the framework employs micro-benchmarks measuring performance metrics of compute and storage services. For the application level, e.g., query operators or full queries, the framework uses our query engine.

To execute experiments, the framework deploys and invokes cloud function binaries. Figure 3 gives an overview of this process. Each experiment defines a configuration, which is submitted to a driver. Depending on the experiment level, the driver invokes a specific function binary on its target platform. For resource-level microbenchmarks, the driver invokes one or more instances of the following cloud functions.

Network I/O. To analyze network performance in isolation, the network measurement function uses iPerf3 [72], an open-source network performance measurement tool. Our function employs the C API, which allows fine-grained parameter tuning. The function sends or receives randomly generated data for a pre-specified time.

Storage I/O. The storage I/O measurement function writes or reads randomly generated files of fixed size and number to or from a storage service. For latency measurements, the function calls the synchronous service APIs. For throughput measurements, it calls the asynchronous APIs from a fixed thread-pool.

Minimal. This binary incorporates the minimum amount of code for a cloud function and is a no-op. It does not link any libraries, but random BLOBs of pre-specified sizes for startup experiments.

For application-level experiments, the driver runs queries with our query engine by calling the query coordinator function, which in turn breaks queries into tasks and schedules worker functions for them. We employ the following suite of queries.

Queries. Our query suite includes TPC-H Q1, Q6, and Q12, as well as TPCx-BB Q3. These queries are I/O-heavy and thus lend themselves well to evaluate cloud resources. We specifically avoid queries that benefit from sophisticated optimization or execution techniques that hide resource aspects. Q1 and Q6 select, project, and aggregate data. Q3 and Q12 are join queries with a broad set of operators, including user-defined functions (UDFs).

Table 3 gives an overview of the experiment configurations. For configurations targeting EC2, we employ a shim layer that resembles the Lambda execution environment to run functions on VM hosts.

When the experiment ends, the driver receives result metrics from multiple sources: Logs, traces, and a response from the invoked function. For resource-level experiments, the metrics include, e.g., timestamps, request counts, latencies, and throughputs. The driver then aggregates these results and estimates the experiment cost using the AWS price list service, disregarding any bulk discounts. For application-level experiments, the query coordinator function returns high-level metrics such as query latency and cost. Finally, the driver stores the results in a JSON file and hands them to a plotter for visualization.

3.2 Query Execution Support

The evaluation framework integrates a serverless query engine to support the execution of end-to-end workloads [48]. This allows to understand how resource effects translate to application performance. The query engine is designed to run entirely on serverless infrastructure. Figure 4 shows an overview of its shared-storage architecture with state kept in serverless storage. The coordinator and worker nodes are deployed as serverless functions and use shared serverless storage to load inputs and communicate outputs. As a deployment alternative, the query engine can run on VMs, enabling the direct comparison of FaaS and IaaS-based execution.

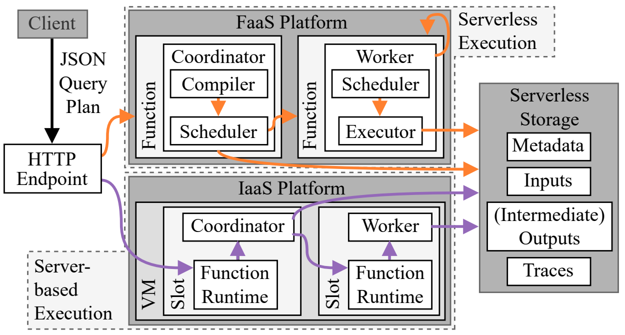

Figure 4: Execution modes of the Skyrise query engine with cloud functions (upper path) and servers (lower path).

To execute a query, the framework's driver sends a physical query plan in JSON format to an HTTP endpoint [34]. On an FaaS platform, this triggers a serverless function running the coordinator. In an IaaS deployment, the request is routed to the same coordinator binary yet running on a provisioned VM with our shim layer. In both cases, the plan is passed to the coordinator. A plan contains pipelines of physical operators as well as the dependencies between the pipelines. The coordinator fetches the metadata on the referenced pipeline input datasets, including the number and sizes of the files. The coordinator then compiles a distributed query plan, deciding on the number of fragments per pipeline for data-parallel execution and on worker sizing. This plan is the same for FaaS and IaaS deployments. Next, the coordinator schedules the pipelines stage-wise based on their dependencies. In serverless execution mode, the scheduler invokes a worker function for each pipeline fragment. In server-based execution, it queues and distributes the fragments across the available worker slots. A worker parses its query fragment and schedules the operators for execution. Workers use a vectorized execution model. The execution includes reading input partitions in batches from shared storage, generating partitioned outputs, and writing them back to storage. Upon completion of the final query pipeline, the coordinator returns a JSON response with the location of the query result in serverless storage, the query runtime and cost. This response is wrapped and sent back to the HTTP client.

To isolate and analyze query subflows, such as distributed scans and shuffles, the query engine supports the injection of synchronization barriers into its execution. This mechanism is implemented as an extra operator that polls a shared queue for a barrier condition.

Moreover, the engine traces runtime information with query context. This information can be compared between distributed workers, as their clocks are tightly synchronized [18, 49].

Finally, the query engine adopts a number of techniques for an execution performance that is representative of other cloud-based data processing systems. They fall broadly into two categories.

Exploiting Serverless Compute Elasticity. To start up a large cluster of workers quickly, the query engine employs a two-level function invocation procedure [93]. Scheduling 256 or more workers, the coordinator parallelizes function calls across a subset of workers. For a query comprising W query fragments, it invokes $\sqrt{W}$ workers, each with a list of $\sqrt{W}$ query fragments that in turn invoke $\sqrt{W} - 1$ workers before executing their own fragment.

Table 3: Overview of experiment configurations.

System under Test	Driver	Functions	Parameters	Metrics
Lambda	FaaS Platform	Minimal, Network I/O, Storage I/O	Instance Size & Count	I/O Throughput, Startup Latency, Idle Lifetime
EC2	IaaS Platform	Network I/O, Storage I/O	Instance Type & Count	I/O Throughput, Startup Latency
S3, DynamoDB, EFS	IaaS & FaaS	Storage I/O	File Size & Count	I/O Throughput, IOPS, Latency
Skyrise Query Engine	Data System	Query Coordinator, Query Worker	Queries, Data Size, Deployment	Query Latency & Cost

To reduce the function invocation latency, we decrease the duration of coldstarts and increase the probability of warmstarts. We keep binary sizes small (< 10 MiB) by linking against the library versions present in Lambda sandboxes and stripping off any unneeded symbols. The deployment artifacts are not specialized towards any query. As such, they can be reused as long as they are cached in the FaaS platform sandboxes.

Efficient Data Access on Serverless Storage. To execute queries efficiently over data stored in columnar file formats on cloud storage, the engine divides large storage requests into smaller chunks to process them in parallel. Straggling requests are retriggered after a size-based timeout. Parquet [96] and ORC [94] file metadata is read to identify relevant data and push down projections and selections.

4 PERFORMANCE EVALUATION

In this section, we present the results of our experiments using our evaluation framework. We give insights into both low-level and workload-specific performance of serverless infrastructure to better understand its potential for data processing. Specifically, we examine the following aspects in the corresponding sections:

- The dominant source of performance variability in serverless function networking (Section 4.2)
- The choice of storage for serverless analytics and the major source of variable performance (Sections 4.3 and 4.4.1)
- The control of the above aspects and their translation to application-level performance (Section 4.5)
- A quantification of the performance variability stemming from temporal and geographical differences (Section 4.6)

4.1 Experimental Setup

We run all experiments on AWS in the availability zone (AZ) us-east-1a [40] in a single virtual private cloud (VPC [19]) network and in the time frame of February to October 2024 unless stated otherwise. For our experiments, we deploy ARM-based Lambda functions and on-demand VMs of the EC2 C6g family of instances[10]. Our driver has low resource requirements and runs continuously on a c6g.xlarge instance across experiments. All other compute resources are newly created for each experiment configuration and repetition. To run our large-scale experiments, we asked AWS to increase our account quotas for parallel function invocations as well as vCPUs in our VM fleet to 10,000 and 5,000 [38]. As serverless storage services, we consider S3, S3 Express [26], DynamoDB, and EFS. We do not consider the deprecated S3 Select service [43]. In our evaluation, we do not include VM-based container services (like the Elastic Kubernetes Service [23]) or provisioned storage options (like ElastiCache [24]). We also do not include the serverless offerings from other cloud providers due to budget constraints.

[10]C6g instances are still much more widely deployed than the more recent C7g family [11, 116] and thus easier to provision in large numbers (>100) for our experiments.

We track service usage via a client hook that counts requests, including failures and retries. Based on the request counts and the runtimes of the employed cloud functions and VMs, we calculate and report the experiment cost. In total, our experiments perform millions of function invocations and billions of storage requests moving hundreds of terabytes of data, adding up to around $4,000.

4.2 Burstable Function Network Bandwidth

Distributed data systems require high network throughput. This is especially true for serverless systems with stateless compute and disaggregated storage, as they scan large amounts of data from remote storage. These systems also shuffle data between nodes via remote storage. Thus, both scanning and shuffling rely on the network. However, the network performance of cloud functions is usually not specified. Cloud vendors neither expose nor guarantee the network bandwidth that functions achieve.

In this experiment, we employ network I/O functions as clients while we deploy iPerf3 servers on EC2 instances with increased network throughput [10, 20], so that they do not become the bottleneck. A single server serves up to 10 clients. A well-known limitation of EC2 is the 5 Gbps limit for single-flow network traffic. To bypass this limitation and explore the full potential bandwidth of functions, we establish multiple paths, i.e., TCP connections, between each pair of endpoints. We utilize functions with 4 vCPUs, and allocate one TCP connection per vCPU. This allows us to measure a theoretical network bandwidth of up to 20 Gbps per function [20].

4.2.1 Network Bursting. We run the network microbenchmark for five seconds, including an intermittent break of three seconds with no traffic. We repeat the experiment ten times and plot the run with the median network throughput. Figure 5 shows that a function initially delivers an inbound bandwidth of 1.2 GiB/s and is capable of maintaining this bandwidth for 250 milliseconds. Afterwards, the bandwidth drops to zero and shows regular spikes in which data is transferred. We observe that this burst is renewable, i.e., it reoccurs after the 3-second break, yet the duration of the second burst is shorter. A similar picture emerges for the outbound bandwidth, although the bandwidth is reduced and shows higher variation. We attribute this to the additional overhead of data generation of the iPerf3 library. We conclude that the inbound and outbound token buckets are maintained independently of each other.

This is consistent with the rate limiting parameters of the virtual machine monitor that Lambda builds on. According to our throughput measurements, both buckets are configured with an initial capacity of ~300 MiB. Once the token bucket empties, 7.5 MiB of data can be consumed in 100 millisecond intervals, resulting in a baseline bandwidth of 75 MiB/s. Furthermore, we find that the token bucket refills halfway to the initial capacity as soon as a function stops utilizing the network or terminates. This implies that a one-off, non-rechargeable budget of ~150 MiB exists in addition to a rechargable bucket capacity of ~150 MiB.

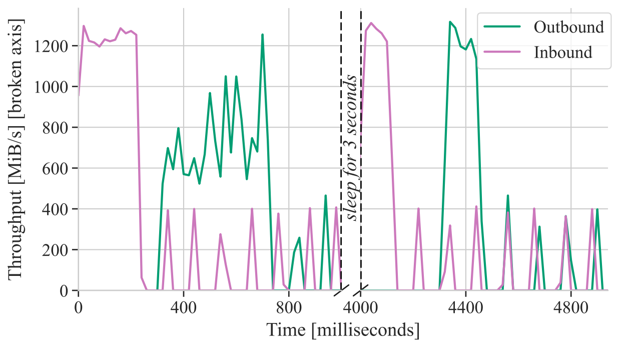

Figure 5: Function network throughput at 20 ms intervals with short sleep to refill inbound/outbound token buckets.

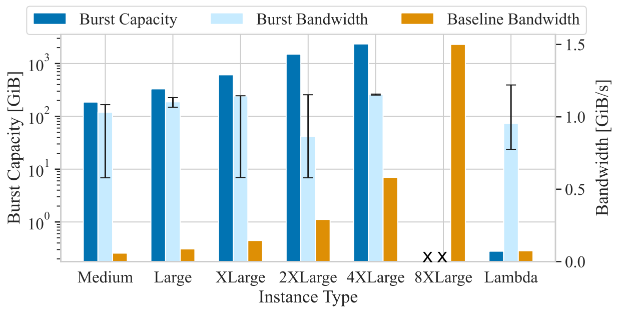

Figure 6: EC2 C6g and Lambda network bursting behavior with burst and baseline throughput, and token bucket size.

We rerun our microbenchmark with EC2 VMs of varying sizes as clients. The benchmark duration depends on the VM size and ranges from three to 45 minutes. We run the experiment three times per configuration and again report the median burst throughput. Figure 6 shows how the network performance of Lambda and EC2 compares. We report the initial capacity of the token bucket for the burst mechanism in GiB. We also report the bandwidth under burst and the sustained baseline bandwidth. Both services employ a bursting mechanism. Lambda allows throughput to burst for a short period of time, while the token bucket size of EC2 instances and the duration of their burst are substantially longer and increases with instance size. We see a high variation for both EC2 and Lambda network burst throughputs, yet very stable burst capacities.

4.2.2 Scalable Network Performance. To study the scaling behavior of Lambda's bursting network performance, we conduct another experiment mapping 32 to 256 functions on a cluster of (4–26) iPerf3 servers. We measure the aggregated network throughput in two different settings. As organizations often deploy their applications in customer-owned VPCs, we restrict parts of the experiment to a VPC within a single AZ. We omit the customer-owned VPC in the second part of the experiment and compare the results of both settings. Figure 7 shows that the baseline and burst bandwidths scale horizontally. We attribute this to the ability of Lambda to place functions effectively. However, we observe limited scalability if the experiment runs in a customer-owned VPC within a single AZ. In particular, we see a hard limit of ~20 GiB/s in throughput. Restricting the service to deploy functions within a VPC appears to either hinder its scheduling flexibility or to introduce a network throughput quota. We conclude that the burstable network throughput of Lambda is significant, deterministic, and scalable (outside of VPCs). Data processing systems should thus aim to exploit it.

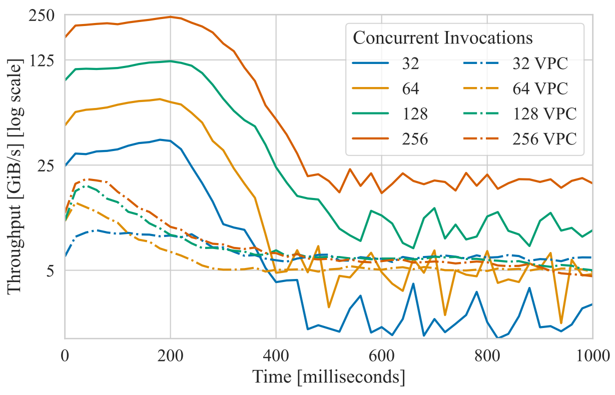

Figure 7: Aggregated network throughput at 20 ms intervals for varying concurrency (32 to 256) with/without VPC.

4.3 Comparing Serverless Storage Options

Object stores, key-value stores, and distributed filesystems are seen to be complementary in their performance characteristics in terms of throughput, IOPS, and latency. To satisfy diverse I/O requirements, data-intensive applications often build on a mix of these options [46, 82]. However, there is little empirical research on when to use which option over another [95, 101] or when none of them are sufficient, leaving a gap in the serverless storage landscape [81, 100]. For this reason, we conduct a comprehensive comparison of the current serverless storage options on AWS. We study the performance and price tradeoffs between the S3 object store, the DynamoDB key-value store, and the network filesystem EFS in the context of large-scale data processing. We evaluate these storage services for throughput of up to hundreds of gigabytes and hundreds of thousands of requests per second. In addition, we analyze their latency distribution over millions of requests.

For these experiments, we employ our storage I/O function. The function runs on EC2 VMs, because EC2 burst capacity allows for sustained high bandwidth throughout the experiments. We use c6gn.2xlarge instances with eight vCPUs, 16 GiB memory, and burstable network bandwidth of up to 25 Gbps. In experiments with distribution, all instances synchronize via a shared queue upon startup to ensure concurrent execution. To reduce storage-side scaling and caching effects, we keep repetition durations short (<5 minutes) and intervals between repetitions long (>12 hours). Unless otherwise noted, we present the median out of three repetitions. In our comparison, we consider the S3 Standard and Express storage classes. We further include DynamoDB with on-demand capacity [8] and strongly consistent reads [7], as well as EFS with elastic throughput and synchronous writes [21]. All services only charge for consumption and provide at least the read-after-write consistency model of S3 [15, 80].

4.3.1 Throughput. Serverless systems scan and shuffle data through storage. Both scanning and shuffling are throughput-heavy operations. In this experiment, we study the scalability of throughput of the storage services with the number of compute nodes generating load. We schedule up to 128 nodes running 32 I/O threads. For S3, we generate and access 64 MiB objects, allowing for roughly 250 GiB/s of throughput on an unpartitioned bucket [32]. For DynamoDB, we employ the largest possible item size of 400 KiB. We access a single table, since sharding over multiple new on-demand tables does not yield higher throughput. We write 4 MiB files to EFS and read them back with no caching.

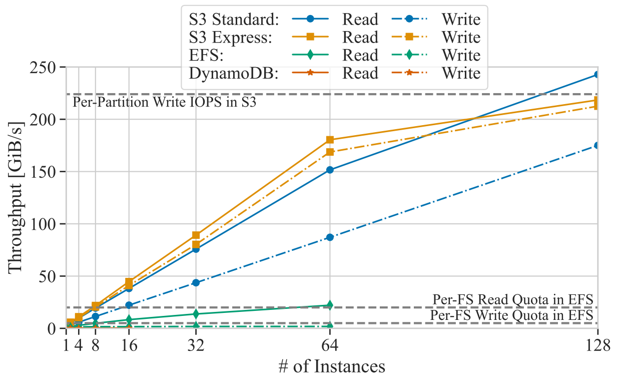

Figure 8: Aggregated read/write throughput of serverless storage services for varying numbers of client VMs.

Our results in Figure 8 show that both the S3 variants scale linearly up to the generated load of ~250 GiB/s. We attribute the difference in their write throughput to the less consistent IOPS performance of standard S3. The serverless versions of DynamoDB and EFS fall short of the target throughput. They each start rejecting requests under contention at different degrees of concurrency. EFS serves up to 64 client VMs with its throughput converging to the quotas (20 and 5 GiB/s [21]) for an individual filesystem instance. DynamoDB's throughput is already saturated by a single client VM and stays at ~380 MiB/s for reading and ~30 MiB/s for writing until most requests get throttled or time out at around 16 clients.

Taking price (cf., Table 2) into account, S3, DynamoDB, and EFS cost 0.00064, 6.55, and 3.00 ¢/GiB/s for reading, respectively. This makes S3 also the by far most cost-efficient option.

4.3.2 Operations per Second. To process queries on large datasets, cloud analytics systems need to access many individual objects or files. This requires to send large numbers of concurrent requests to the storage services. We measure the IOPS performance of S3, DynamoDB, and EFS on newly created buckets, tables, and filesystems. We again run up to 128 nodes, each with 32 dedicated threads sending 1 KiB requests for a total of >250K requests per second. For DynamoDB, we asked AWS to increase the table and account-level IOPS quotas to 250K. Our results are shown in Figure 9. The standard S3 performance is just above the target IOPS for an individual prefix partition [32] with 8K reads and 4K writes per second. S3 Express is not subject to the partition quota, providing the highest IOPS in our comparison with 220K for reads and 42K for writes. DynamoDB also provides slightly more IOPS than defined by the quotas for new on-demand tables [8], with 16K read IOPS and 9.6K write IOPS. We miss the per-filesystem quotas of EFS by more than an order of magnitude, despite closely following the documentation [21]. The read IOPS double via sharding over two filesystems, but do not scale further.

4.3.3 Latency. Compared to compute local storage, disaggregated cloud storage entails orders of magnitude higher latency. While analytics are usually throughput-bound, latency is still important and compensated via increased request size or concurrency [61, 93]. To determine the latency distributions of S3, DynamoDB, and EFS, we send one million 1 KiB read and write requests to every service. To keep the experiment duration moderate and the load on the services low, we employ 10 clients using the synchronous APIs [17]. We present our results in Figure 10.

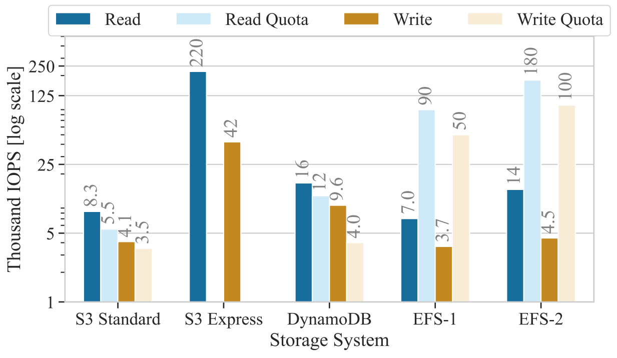

Figure 9: Operations per second and container-level quotas for each serverless storage system. For EFS, we present configurations with one (EFS-1) and two (EFS-2) filesystems.

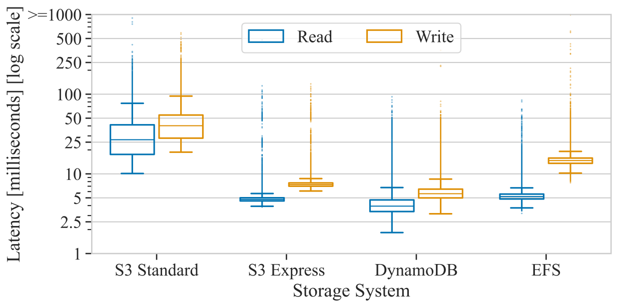

Figure 10: Latency distribution of each serverless storage system for one million read/write requests.

We observe that S3 Standard has the highest median (27 ms for reads and 40 ms for writes) and tail latencies. Out of 1M read requests, 95% completed in 75 ms and the slowest requests took just over 10 s (374X of the median). S3 Express benefits from its zonal deployment [26] and provides significantly lower and less variable latencies with the median and 95th percentile read latencies at around 5 ms. DynamoDB exhibits slightly lower yet more variable latencies than S3 Express. Finally, EFS provides similarly low and consistent read request latencies as S3 Express and DynamoDB, but shows 2–3× higher write latencies.

4.3.4 Choosing Storage for Data Processing. Our results allow us to differentiate between the available serverless storage options. The option that provides the most economic scalable throughput is S3. Standard S3, however, offers the lowest out-of-the-box IOPS performance at the highest request latency. For these reasons, the recent S3 Express variant is an attractive alternative. S3 Express offers the highest IOPS throughput at consistent low latency, but at higher cost. DynamoDB provides the lowest latency, yet also the lowest throughput. Finally, EFS shows a balanced performance, but it is inferior to S3 Express in every evaluated dimension at a higher price point. We conclude that S3 is the most suited option for scalable data processing and focus the rest of our evaluation on methods based on object storage.

4.4 Object Storage IOPS Scaling

Object stores provide scalable throughput at request latencies that are acceptable for many analytical workloads [53, 107]. One major performance limitation of object stores is their low default IOPS, making them reject requests under spiking load. This is problematic for concurrent query workloads on the same datasets. Similarly, it is a challenge to serverless data analytics systems that shuffle intermediate data through object storage [100]. Every

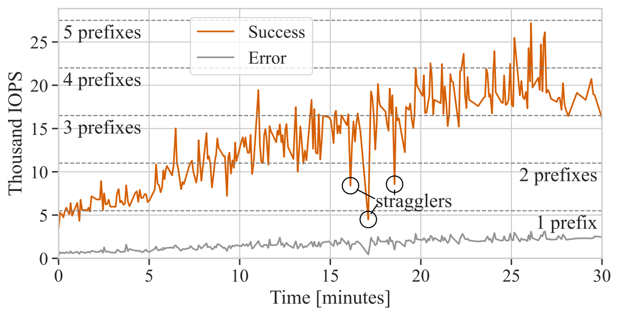

Figure 11: S3 IOPS scaling from one to five prefix partitions.

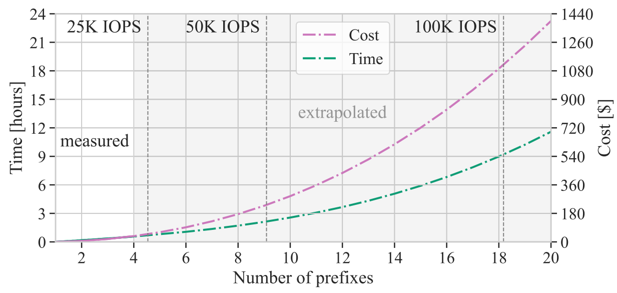

Figure 12: Required time and budget for S3 IOPS scaling.

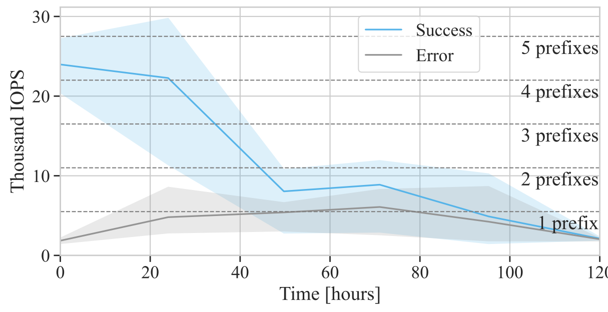

Figure 13: S3 scaling down from five to one prefix partitions under hourly and daily load patterns.

serverless worker node has to read all relevant columns of all assigned partitions of all workers in the preceding stage. For terabyte-scale queries, even advanced shuffle strategies [93, 98] require thousands of requests. In this experiment, we show how to scale up IOPS in S3 both reliably and efficiently. We also describe how S3 scales down partitions and IOPS when idle.

In the S3 object store, user data is horizontally partitioned on the object key namespace. Objects are organized in string key prefixes, which can span from an entire bucket to an individual object [39]. The prefixes are backed by physical partitions on S3 storage nodes and serve 3.5K writes and 5.5K reads per second [32]. To account for workload changes, prefixes are split and merged automatically and gradually over time (cf. Section 2.2).

4.4.1 IOPS Scaling. We examine the fraction of successful requests under carefully controlled increasing load to understand object storage IOPS scaling. This is necessary, because S3 throttles requests quickly when load spikes [32, 100]. For this experiment, we reuse the microbenchmark from the previous section with Lambda for compute and S3 for storage. The S3 client is configured with a request timeout of 200 ms for retries and exponential backoff [50]. This results in an eager but not aggressive retry behavior. We start with 20 Lambda instances that each read one thousand 1 KB objects concurrently. An instance has four vCPUs and generates ~250-350 asynchronous requests per second, so the overall cluster saturates an S3 partition (with ~5–7K IOPS). We run 10 repetitions with this configuration. Then, we continue to run more configurations the same way, each incrementally adding two cluster instances (and ~600 IOPS load) up to a total of 100 instances and around 30K requests per second.

The results are depicted in Figure 11. We plot the average successful and failed (throttled or timed out) read operations per second for each repetition over time. Our results show that S3 scales nearly linearly from ~5–27.5K IOPS with this load pattern. While scaling out, IOPS performance has a high variance with a relative standard deviation of up to 29% for individual configurations. In addition, we observe three significant performance drops about 16–19 minutes into the experiment. Although the overall error rate is constant at just above 10% throughout the experiment, few S3 clients see their requests repetitively being rejected. These clients then wait exponentially longer after every attempt and turn into stragglers in their respective repetition. Hence, the drops in IOPS are due to our client configuration and not S3's scaling behavior.

In our experiment, we observe S3 scaling from one partition serving 5.5K IOPS to five partitions providing 27.5K IOPS. This process takes about 26 minutes and 63 million requests costing $25. To determine what it would take to make S3 partition our prefixes further for higher IOPS performance, we extrapolate time and cost based on our measurements. In Figure 12, we show our measured and extrapolated data points for up to 20 prefix partitions totaling 110K IOPS. Given our load pattern and polynomial fitting method, we see that it would take 2 hours and $228 to reach 50K IOPS. Further, it would require 9 hours and $1,094 to get 100K IOPS. This makes IOPS scaling a quickly growing expense for users while S3 only allocates resources linearly and with delay as a form of admission control (cf. Section 2).

4.4.2 Downscaling Behavior. After scaling up IOPS in S3, we study the process of scaling back down in periods of low load. This is to better understand when S3 begins to throttle requests and merge prefix partitions. We conduct this experiment in direct succession to the experiment in Section 4.4.1. We first wait for an extended period of time before we run three repetitions of the last (and largest scale) employed configuration of the storage microbenchmark. We repeat this procedure until performance drops down to the level of a single partition and stays there. Since our experiment generates load against S3, it potentially influences its own outcome. There is a tradeoff between the frequency and accuracy of the measurements. For this reason, we run the experiment on two separately scaled buckets with hourly and daily measurement intervals, respectively. In Figure 13, we see the results of scaling down our S3 buckets from five partitions to one. We plot the band of IOPS across all three repetitions per interval for each series of measurements. We take the highest IOPS per interval as an indication for the number of the remaining partitions in the buckets. Our results indicate that the overall downscaling process takes between four and five days. After a full day of inactivity, all five partitions remain available to serve load. Two out of the five partitions continue to be available for an additional three days before IOPS performance returns to the level of a single partition. Since IOPS performance scales linearly and remains high over extended periods of low load, we conclude that IOPS scaling is a relevant optimization for analytical workloads, even if they are infrequent (with hourly or daily load patterns).

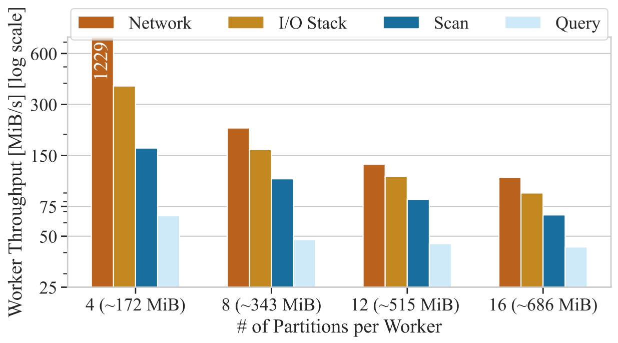

Figure 14: Query worker throughput for given input sizes within and beyond network burst budget with TPC-H Q6.

4.5 Exploiting Serverless Characteristics

We now study how the described performance characteristics of serverless resources translate to data-intensive applications. Since the translation is complex, we first show the impact on data system components and then on full analytical queries. In the following experiments, we use queries and datasets from the TPC-H [108] and TPCx-BB [109] benchmarks. We run all queries on the tables of scale factor 1.000. The tables are partitioned into Parquet files and stored on S3. We employ the standard generators and do not partition or sort on any specific keys. We provide details in Table 4. We run the queries on the Skyrise query engine. Query workers have 4 vCPUs and 7.076 MiB RAM.

4.5.1 Network Bursting for Scan-heavy Queries. In our network analysis, we determine a budget of 300 MiB for unthrottled throughput, which benefits throughput-heavy tasks like table scanning. We show the benefit of burst-awareness by running the scan query TPC-H Q6. We assign workers an increasing number of partitions, gradually exceeding their budgets. We present our results in Figure 14. We plot the expected throughput per worker according to our network model and the actual throughput of the Skyrise engine's I/O stack, the scan operator, and the complete query. We see the performance impact of S3 request handling, decompression and deserialization, and the scan and full query logic. Across the partitioning settings, we see that queries fully exploiting the network burst are up to 53% faster. We derive that serverless data systems benefit from calibrating and managing the network ingress/egress of their compute nodes.

4.5.2 IOPS Scaling for Queries with Shuffles. Our analysis of object storage shows that IOPS performance scales under sustained load. We replicate and exploit this behavior at the query level. We run the I/O-heavy TPC-H Q12 join query with 320 workers. At this degree of parallelism, the shuffling for the join requires about 42.000 read operations and is constrained by default rate limiting. For shuffling, we employ three different storage setups. We use a new S3 Standard bucket, another bucket that has just been used for query execution for 15 minutes, and an S3 Express bucket. Our results are shown in Figure 15. For reference, we plot the read IOPS throughput for all setups, as measured in Sections 4.3.1 and 4.3.2. We see that the runtime of the shuffle and the entire query are generally reduced by about 50% and 20%, respectively. While scaling IOPS of object storage takes too long to do as part of interactive queries, throughput should be considered when planning query parallelism.

Table 4: Datasets used in the experiments. Partition sizes are the mean Parquet file size with ZSTD compression.

TPC Table	@ SF1000	Partitions	
	Size [GiB]	#	Size [MiB]
H-Lineitem	177.4	996	182.4
H-Orders	44.9	249	176.1
BB-Clickstreams	94.9	1,000	92.7
BB-Item	0.08	1	75.8

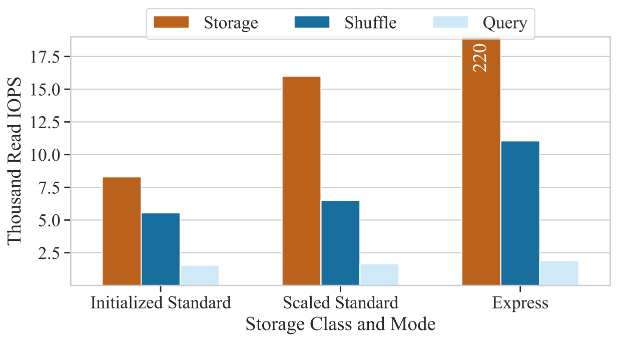

Figure 15: IOPS throughput of various S3 classes and their performance impact on TPC-H Q12 and its shuffle.

4.6 Query Performance Variability

Now that we better understand network bursting in serverless functions and object storage scaling, we quantify the impact of geographical, temporal, and other local aspects on the variance of query performance. For this set of experiments, we add two queries that implement scan-heavy aggregation (TPC-H Q1) and I/O-bound MapReduce jobs (TPCx-BB Q3). We specifically use these queries on synthetic data to avoid data and computational skew. We further ensure that the Skyrise query workers stay within their network burst budgets and consistently use either cold or warm function instances and storage buckets. We deploy and run our query suite in the AWS regions us-east-1, eu-west-1, and ap-northeast-1. We conduct one experiment by running our query suite over a workday with 15 minute intervals between runs, dubbed cold. We then carry out a second experiment with the queries run with no wait time, i.e. warm, over three hours.

We present our results in Table 5. We report two metrics, namely the median to base median ratio (MR) and the coefficient of variation (CoV). MR normalizes the query suite runtime with the median of the us-east-1 (US) region. We use CoV [102] as a measure of variation within a region. We see a mixed picture. The variance is low across the US and AP regions, but significant compared to the EU (~50%). In the EU, the start of large function clusters takes significantly longer, likely due to contention within the region. For the US and AP regions, the local variability is higher, with the cold experiment showing yet higher variance than the warm one. We deduce that more frequent usage leads to pre-provisioning of resources and thus to more robustness. Localized factors continue to considerably impact variability in performance, necessitating further examination.

Table 5: Performance variability between and within regions in short experiments and over a weekday.

Measure	US	EU	AP
Cold MR (US)	1	1.48	0.95
Cold CoV (24h)	22.65	4.76	7.65
Warm MR (US)	1	1.52	0.96
Warm CoV (3h)	5.23	8.96	6.44

5 ECONOMIC VIABILITY

In this section, we discuss the economic implications of building data processing systems on serverless infrastructure. We explain the assumptions for our discussion in Section 5.1. We then examine the cost-saving potential of FaaS-based query execution in Section 5.2. In Section 5.3, we determine the break-even points for scanning and shuffling data on serverless storage.

5.1 System Architecture and Cloud Pricing

In our examination, we assume a distributed architecture that disaggregates both persistent and ephemeral storage and uses small and stateless compute nodes. This is the inherent architecture of serverless data processors [93, 98] and also the target design for some commercial cloud systems [86, 112]. While serverless systems use this architecture due to the restrictions discussed in Section 2, industrial systems adopt it for elasticity. Making this assumption, we factor out the inefficiencies of disaggregation and distribution [77, 104] compared to monolithic and single-node systems, which can cache and process data entirely in memory.

Beyond this, we assume the current service pricing models of AWS (cf. Tables 1 and 2, [9]). They are comparable to those of Microsoft Azure [91] and GCP [68] and are reasonably stable over time [31].

5.2 Breaking Even with Serverless Compute

FaaS platforms have higher compute unit prices and performance overheads [103] than IaaS platforms. In return, they offer automatic, elastic, and fine-grained scalability. In this section, we study the economic tradeoff of IaaS and FaaS deployment of data processing systems. We determine the performance overhead and cost of FaaS-based execution for selected queries. In addition, we identify the cost-savings potential enabled by elasticity.

For this experiment, we rerun our query suite from Section 4.6 in two configurations. We first run the queries on Skyrise in Lambda with each function having 4 vCPUs and 7.076 MiB RAM. Then, we deploy Skyrise on a cluster of 284 EC2 C6g.xlarge VMs with 4 vCPUs and 8 GiB RAM. The functions are warmed up and the VMs are started before the experiment begins. For both configurations, the query plans and physical resources are the same. The Skyrise workers employ S3 to read the base tables and shuffle intermediate results. We run the query suite ten times each and collect statistics from the run with the median runtime. The statistics include the runtime, the accumulated function lifetime, and the number and size of the storage requests per query. We present our results for the queries TPC-H Q6 and Q12 in Table 6.

Table 6: Execution statistics and derived economic metrics: Break-even FaaS query throughput for peak-provisioned IaaS and intra-query peak-to-average node ratio.

Query	H-Q6	H-Q12
IaaS Runtime [s]	5.2	18.1
IaaS Hourly Cost [$/h]	27.34	38.62
Storage Requests	1,401	30,033
Shuffle I/O Size [KiB]	0.4	1.1–2,078
Query I/O Cost [¢/Q]	0.16	1.39
FaaS Runtime [s]	5.7	19.2
Cumulated Time [s]	515.9	2,227.3
FaaS Query Cost [¢/Q]	4.87	21.19
Break-Even [Q/h]	*558*	*182*
Peak-to-Average-Nodes	*2.21×*	*2.43×*

Table 7: Pricing and performance of AWS server storage. SSD (NVMe) and EBS (gp3) numbers are of 950 GiB volumes.

RAM Price [¢/MiB/h]	0.00022
SSD Price [¢/h]	7.04
SSD Throughput [MiB/s]	2,000
SSD IOPS	215,000
EBS Price [¢/h]	21.88
EBS Throughput [MiB/s]	593.75
EBS IOPS	16,000

Query Runtime Slowdown. In the FaaS deployment, the end-to-end latencies for Q6 and Q12 are 10% and 6% higher. The primary reason is the startup time of the functions for every query stage compared to no startup overhead in the IaaS deployment with pre-provisioned VMs. In addition, there are occasional cold start stragglers, in particular for the coordinator. These stragglers do not impact cost, because other functions do not idle waiting for them.

Query Cost and Break-Even Throughput. We calculate the FaaS cost of a query based on the aggregated lifetimes of both the coordinator and worker functions in all stages. We relate the query cost to the cost of a peak-provisioned VM cluster to determine the break-even throughput. A C6g.xlarge instance costs 0.136 $/h. The peak number of instances used for Q6 is 201 and for Q12 is 284. Thus, FaaS deployment is economical for up to 558 runs per hour of Q6 or 128 runs of Q12. For adaptively provisioned clusters with higher utilization, the break-even throughput decreases proportionally.

Intra-Query Elasticity. Analytical queries consist of multiple stages that may have very different input sizes and computational requirements. Skyrise schedules 283 nodes in the first stage of Q12 to scan and filter 222.3 GiB and a single node to aggregate 105.5 KiB (six orders of magnitude smaller) into the final result in the last stage (cf. Figure 16). We calculate the peak-to-average node ratio across stages as potential cost-savings factor compared to static peak provisioning for queries. For Q12, this ratio is 2.43×.

5.3 Breaking Even with Serverless Storage

Cloud functions can neither cache data beyond their short lifetimes nor can they communicate directly. Thus, FaaS-based query workers need to access remote cloud storage to scan and exchange data. This section studies the cost implications of these limitations. In our discussion, we exclude performance concerns and assume scans overlap with computation and shuffles are throughput-bound.

5.3.1 Reads in the Cloud Storage Hierarchy. We build our caching discussion on Gray's regularly revisited five-minute rule for trading off memory and disk accesses [44, 71]. We introduce two variations of the rule to account for different cloud storage pricing models. We use the first variant for cloud storage that is priced by capacity only, such as VM-based RAM and SSDs, as well as network drives (cf. Table 7). We calculate the break-even interval (BEI) in seconds as follows.

$$BEI = \frac{PagesPerMB}{AccessesPerSecondPerDisk} \times \frac{RentPerHourPerDisk}{RentPerHourPerMBofRAM}$$

The second variant reflects pricing by the number of requests, as in serverless object storage and key-value stores.

$$BEI = PagesPerMB \times \frac{PricePerAccessToTier2}{RentPerSecondPerMBofTier1}$$

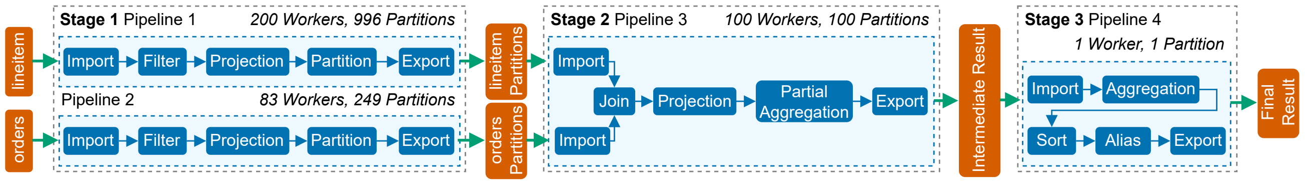

Figure 16: Skyrise query execution plan for TPC-H Q12.

Table 8: Break-even intervals for different data access sizes and storage combinations in AWS (us-east-1) in July 2024.

Access Size	4 KiB	16 KiB	4 MiB	16 MiB
RAM/SSD	38s	31s	31s	31s
RAM/EBS	27min	7min	3min	3min
RAM/S3 Standard	2d	12h	3min	41s
RAM/S3 Express	23h	6h	36min	39min
RAM/S3 X-Region	2d	20h	9h	9h
SSD/EBS	13h	3h	1h	1h
SSD/S3 Standard	59d	15d	1h	21min
SSD/S3 Express	29d	7d	18h	20h
SSD/S3 X-Region	70d	26d	11d	11d

For our comparison along the hierarchy of cloud storage for query workers, we include RAM, SSDs, EBS network drives, and S3 object storage. These are the established options for cloud data systems [107]. We assume workers to run on EC2 C6gd VMs with NVMe SSDs [12]. We further assume on-demand prices. Lower reserved prices would increase the break-even intervals proportionally, whereas higher Lambda prices would decrease them. Table 8 shows the results of our calculations. We draw the following conclusions.

Relevance of SSDs. The break-even for RAM versus SSD with 4 KiB accesses is 38s, one order of magnitude less than a decade ago [44]. This is due to increased IOPS performance and decreased prices. The interval for larger accesses does not get shorter because the SSD bandwidth in EC2 of 2 GiB/s [83] becomes the bottleneck, limiting the SSD IOPS in above formula. Conversely, the break-even for SSD and object storage is hours to days for all but very large (≥16 MiB) accesses. As a result, caching on SSD is economical for a wide range of access sizes and frequencies. This explains the prevalence of SSD caches in VM-based systems [46, 59]. While cloud functions support SSDs, caches are bound to the short lifetime of seconds to minutes at most, limiting their cost effectiveness. Cloud functions do not support network drives, which potentially outlive them.

Analytics on Cold Data. Query workers do not benefit from SSD caches when accesses occur at most on an hourly basis and are in the megabyte scale, e.g., one 4 MiB access per hour. Furthermore, these caches may have low hit rates, as workers lose their state when they are removed in idle times and are readded later. This resembles the cold data analytics workload that serverless systems target [93].

Pricing Model. RAM/SSD break-even intervals are constant within an EC2 instance family (e.g., C6g). This is due to SSD IOPS performance growing with its instance and price, and both parameters being on opposite sides of above equation. Data transfer fees, as for S3 Express and cross-region access, invalidate the initial rule that the break-even is inversely proportional to the access size.

Table 9: Break-even data access sizes for different instance types and storage systems in AWS (us-east-1) in July 2024.

Instance Type	ElastiCache on-demand	C6g on-demand	C6gn on-demand	C6gn reserved
S3 Standard	0.9 MiB	2 MiB	6.5 MiB	15.1 MiB
S3 Express	–	–	–	–

5.3.2 *Impact of Shuffle I/O Size.*

To shuffle intermediates in a serverless system, every worker reads its respective partition(s) from every object of the preceding stage(s) from object storage. For large queries, the cost of the resulting read requests dominate the overall query cost. For this reason, many systems employ key-value stores on provisioned VM clusters [81, 97, 100] to shuffle intermediates. The shuffle capacity of a cluster is the aggregated network throughput of the VMs and its cost is the combined cost of the VMs. Since object storage requests are priced independently of their size, there is a break-even access size at which object storage becomes more economical for shuffling. We calculate this access size (BEAS) in MB as follows.

$$BEAS = PricePerAccess \times \frac{MBPerHourPerServer}{RentPerHourPerServer}$$

Table 9 shows our results for different cluster VM types from EC2 and ElastiCache, as well as the S3 Standard and Express storage classes. The EC2 VMs are from the C6g family, including the network-optimized C6gn variant with four times the network throughput at both on-premise and reserved pricing [13]. We derive the following insights.

Case for Large Accesses. Object storage is the cheaper shuffle medium when average accesses are larger than ~1–15 MiB, depending on the VM type and pricing model. In distributed query execution, intermediates are highly partitioned and individual I/Os tend to be small. In our query experiments, they are ~1 KiB–2 MiB (cf. Table 6). There, however, is a range of techniques to increase I/O sizes, including write combining and staged shuffling [93, 98].

Price of VM-based Services. Even basic key-value stores, such as ElastiCache have 2× higher prices than the EC2 VMs that they run on. There is additional potential for workload-optimized VMs and long-term pricing models. This motivates the construction of shuffle systems on self-managed VMs. For S3 or another serverless service to be competitive for small accesses, request prices need to be orders of magnitude lower (cf. Table 2).

Pricing Model. The break-even access sizes are constant within EC2 and ElastiCache VM families, since network throughput grows proportionally with VM size and price. The S3 Express storage class never breaks even with VM clusters due to its data transfer cost component.

6 DISCUSSION

In this section, we present key takeaways from our evaluation of the performance and cost efficiency of serverless infrastructure.

Serverless Performance. Our results show that the key techniques of serverless systems (e.g., tenant isolation and placement, as well as rate limiting) have a large impact on performance. We identify four aspects that have not yet been studied in detail.

(1) **Network rate limiting:** We show that cloud functions and VMs are subject to network bandwidth limiting once they consumed a deterministic burst capacity. Thus, their ingress/egress should be aligned to maximize bandwidth. The accelerated bandwidth can benefit scan-heavy queries.
(2) **Storage IOPS scaling:** Object storage is the most suited option for serverless data analysis. We demonstrate that the strict request rates of object storage deterministically increase under sustained load and decrease in extended idle periods. This process can accelerate join queries.
(3) **Variability factors:** Regional variance can be substantial, but generally temporal variance is higher. More frequent usage leads to pre-provisioning of resources and increased robustness. Substantial sources of local variability remain for further investigation.
(4) **Security conflicts:** Virtual network partitions (e. g., VPCs commonly used in production settings) currently hinder the elasticity of serverless networks significantly.

Serverless Economics. We observe that unit prices are higher in serverless systems, which provision resources on behalf of the users to provide elasticity. We see four ways to optimize cost.

(1) **Infrequent and peak usage:** Users pay for consumed resources only and benefit when workloads are infrequent or peak unpredictably. The serverless pricing model should be combined with cost models for provisioned resources to handle workloads with substantial base load.
(2) **Intra-job elasticity:** Analytical queries and machine learning pipelines have varying resource demands and benefit from intra-job elasticity enabled by serverless compute.
(3) **Economic caching:** Our examination suggests that cold (hourly accessed) data should be kept in object storage and fetched in megabyte-scale granularity. Warmer data should be cached on VM-based SSDs.
(4) **Economic shuffling:** In highly distributed environments, both serverless and provisioned storage services are suboptimal for data shuffling. Users should consider building their own shuffle systems on network-optimized VMs with discounted long-term pricing.

Transaction Processing. The request latencies and prices of current services for disaggregated storage are inadequate for fine-grained, high-throughput operational workloads. Users need to build storage subsystems for efficient transaction processing.

Generality of Results. We believe that our results have a wide relevance, since the studied performance effects are present in VMs and object storage, which are the major building blocks for modern commercial data analysis systems. These systems further start to adopt serverless compute resources for functionality, such as UDFs and ETL. We acknowledge that concrete numbers may change between major cloud providers, over time, and between different geographies. We offer our open-source tooling and methods to validate and revalidate our results.

7 RELATED WORK

This section summarizes related work on serverless infrastructure and data processing systems.

Analysis of Serverless Infrastructure. Prior work includes benchmark frameworks and studies for serverless systems from all major cloud providers [85, 99, 103, 113]. On the application level, they focus on web, IoT, and media applications that require little coordination and state [58, 70, 78, 118]. On the resource level, they cover aspects including the performance and isolation of cloud function CPUs, memory, and disk storage. They provide insights into FaaS platform overheads and scalability. There has been evaluation of the network characteristics of VMs [102, 111] and serverless functions [93, 114]. Serverless storage has been studied in [61, 81, 95, 100, 101]. An additional area of research focuses on performance variability within IaaS platforms [102, 111] and FaaS platforms [98, 110].

We instead evaluate the performance of serverless resources for large-scale and stateful applications. We perform a detailed analysis in AWS running millions of serverless functions and billions of storage requests moving hundreds of terabytes of data. We characterize the burstable network performance of serverless functions and the scalability of various serverless storage systems, including the recent S3 Express. We demonstrate that these properties translate to application performance and quantify the remaining sources of performance variability. We present cost break-even points for serverless compute and storage in the data processing context.

Serverless Data Systems and Applications. Recently, there have been several system prototypes to explore the viability of serverless infrastructure for data processing. Some support general-purpose, MapReduce-style processing [57, 73, 75, 79, 84, 100], and some are SQL query execution engines [47, 93, 97, 98]. Other works evaluate serverless resources for different I/O-intensive workloads, such as machine learning training [55, 74]. Most systems are closed-source, and none allow for the analysis of the impact of resource-level properties on system components and full queries.

In contrast, Skyrise is open-source and allows to explore serverless infrastructure properties across the stack with a suite of microbenchmarks and an integrated serverless query engine.

8 CONCLUSION

We perform an in-depth analysis of serverless network, storage, and compute behavior for data processing in an extensive series of large-scale cloud experiments. Our results provide a detailed understanding of network performance bursting and I/O warming and their influence on query processing. Using our analysis framework, Skyrise, we execute full queries and compare serverless and VM-based execution. We derive several cost break-even points to determine when serverless query processing is economical.

ACKNOWLEDGMENTS

We thank the Master's students that worked on the Skyrise project and implemented components of the evaluation framework: Lars Jonas Bollmeier, Fabian Engel, Tobias Maltenberger, and many more. We also thank the AWS Lambda engineering team for insightful discussions. This work was partially funded by SAP, the AWS Cloud Credit for Research Program, the German Research Foundation (414984028), and the European Union's Horizon 2020 research and innovation programme (957407).

REFERENCES

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *NSDI*. USENIX, 419–434.

[2] Alibaba. 2024. Function Compute. https://www.alibabacloud.com/product/function-compute/. Accessed: 2024-10-07.

[3] Alibaba. 2024. What Is Serverless Computing? https://www.alibabacloud.com/knowledge/what-is-serverless/. Accessed: 2024-10-07.

[4] Amazon. 2018. Announcing Amazon DynamoDB On-Demand. https://aws.amazon.com/about-aws/whats-new/2018/11/announcing-amazon-dynamodb-on-demand/. Accessed: 2024-10-07.

[5] Amazon. 2021. Overview of Data Transfer Costs for Common Architectures. https://aws.amazon.com/blogs/architecture/overview-of-data-transfer-costs-for-common-architectures/. Accessed: 2024-10-07.

[6] Amazon. 2022. Announcing Amazon EFS Elastic Throughput. https://aws.amazon.com/blogs/aws/new-announcing-amazon-efs-elastic-throughput/. Accessed: 2024-10-07.

[7] Amazon. 2023. Amazon DynamoDB Read Consistency. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html. Accessed: 2024-10-07.

[8] Amazon. 2023. Amazon DynamoDB Read/Write Capacity Mode. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadWriteCapacityMode.html#HowItWorks.OnDemand. Accessed: 2024-10-07.

[9] Amazon. 2023. Amazon EBS Pricing. https://aws.amazon.com/ebs/pricing/. Accessed: 2024-10-07.

[10] Amazon. 2023. Amazon EC2 C6g Instances. https://aws.amazon.com/ec2/instance-types/c6g/. Accessed: 2024-10-07.

[11] Amazon. 2023. Amazon EC2 C7g Instances. https://aws.amazon.com/ec2/instance-types/c7g/. Accessed: 2024-10-07.

[12] Amazon. 2023. Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/. Accessed: 2024-10-07.

[13] Amazon. 2023. Amazon EC2 Pricing. https://aws.amazon.com/ec2/pricing/. Accessed: 2024-10-07.

[14] Amazon. 2023. Amazon S3. https://aws.amazon.com/s3/. Accessed: 2024-10-07.

[15] Amazon. 2023. Amazon S3 Data Consistency Model. https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html#ConsistencyModel. Accessed: 2024-10-07.

[16] Amazon. 2023. AWS Lambda. https://aws.amazon.com/lambda/. Accessed: 2024-10-07.

[17] Amazon. 2023. AWS SDK for C++. https://github.com/aws/aws-sdk-cpp/. Accessed: 2024-10-07.

[18] Amazon. 2023. Set the time for your Linux instance. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/set-time.html. Accessed: 2024-10-07.

[19] Amazon. 2023. What is Amazon VPC? https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html. Accessed: 2024-10-07.

[20] Amazon. 2024. Amazon EC2 Instance Network Bandwidth. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-network-bandwidth.html. Accessed: 2024-10-07.

[21] Amazon. 2024. Amazon EFS Performance. https://docs.aws.amazon.com/efs/latest/ug/performance.html. Accessed: 2024-10-07.

[22] Amazon. 2024. Amazon EFS Pricing. https://aws.amazon.com/efs/pricing/. Accessed: 2024-10-07.

[23] Amazon. 2024. Amazon Elastic Kubernetes Service. https://aws.amazon.com/eks/. Accessed: 2024-10-07.

[24] Amazon. 2024. Amazon ElastiCache. https://aws.amazon.com/elasticache/. Accessed: 2024-10-07.

[25] Amazon. 2024. Amazon Kinesis. https://aws.amazon.com/kinesis/. Accessed: 2024-10-07.

[26] Amazon. 2024. Amazon S3 Express One Zone Storage Class. https://aws.amazon.com/s3/storage-classes/express-one-zone/. Accessed: 2024-10-07.

[27] Amazon. 2024. Amazon S3 Pricing. https://aws.amazon.com/s3/pricing/. Accessed: 2024-10-07.

[28] Amazon. 2024. Amazon Simple Queue Service. https://aws.amazon.com/sqs/. Accessed: 2024-10-07.

[29] Amazon. 2024. AWS Lambda Customer Case Studies. https://aws.amazon.com/lambda/resources/customer-case-studies/. Accessed: 2024-10-07.

[30] Amazon. 2024. AWS Lambda Pricing. https://aws.amazon.com/lambda/pricing/. Accessed: 2024-10-07.

[31] Amazon. 2024. AWS News Blog Category: Price Reduction. https://aws.amazon.com/blogs/aws/category/price-reduction/. Accessed: 2024-10-07.

[32] Amazon. 2024. Best Practices Design Patterns: Optimizing Amazon S3 Performance. https://docs.aws.amazon.com/AmazonS3/latest/userguide/optimizing-performance.html. Accessed: 2024-10-07.

[33] Amazon. 2024. Best Practices for Designing and Architecting with DynamoDB. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/best-practices.html. Accessed: 2024-10-07.

[34] Amazon. 2024. Invoking Lambda Function URLs. https://docs.aws.amazon.com/lambda/latest/dg/urls-invocation.html. Accessed: 2024-10-07.

[35] Amazon. 2024. Lambda Function Scaling. https://docs.aws.amazon.com/lambda/latest/dg/lambda-concurrency.html. Accessed: 2024-10-07.

[36] Amazon. 2024. Lambda Instruction Set Architectures (ARM/x86). https://docs.aws.amazon.com/lambda/latest/dg/foundation-arch.html. Accessed: 2024-10-07.

[37] Amazon. 2024. Lambda Isolation Technologies. https://docs.aws.amazon.com/whitepapers/latest/security-overview-aws-lambda/lambda-isolation-technologies.html. Accessed: 2024-10-07.

[38] Amazon. 2024. Lambda Quotas. https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html. Accessed: 2024-10-07.

[39] Amazon. 2024. Organizing Objects Using Prefixes. https://docs.aws.amazon.com/AmazonS3/latest/userguide/using-prefixes.html. Accessed: 2024-10-07.

[40] Amazon. 2024. Regions and Zones. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html#concepts-availability-zones. Accessed: 2024-10-07.

[41] Amazon. 2024. Serverless Data Storage Options. https://docs.aws.amazon.com/whitepapers/latest/serverless-multi-tier-architectures-api-gateway-lambda/serverless-data-storage-options.html. Accessed: 2024-10-07.

[42] Amazon. 2024. Serverless on AWS. https://aws.amazon.com/serverless/. Accessed: 2024-10-07.

[43] Amazon. 2025. Querying data in place with Amazon S3 Select. https://docs.aws.amazon.com/AmazonS3/latest/userguide/selecting-content-from-objects.html. Accessed: 2025-01-06.

[44] Raja Appuswamy, Renata Borovica-Gajic, Goetz Graefe, and Anastasia Ailamaki. 2017. The Five-minute Rule Thirty Years Later and its Impact on the Storage Hierarchy. In *VLDB ADMS Workshop*, Rajesh Bordawekar and Tirthankar Lahiri (Eds.). 1–8.

[45] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andy Konwinski, Gunho Lee, David A. Patterson, Ariel S. Rabkin, Ion Stoica, and Matei A. Zaharia. 2009. *Above the Clouds : A Berkeley View of Cloud Computing*. Technical Report UCB/EECS-2009-28. EECS Department, University of California, Berkeley.

[46] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *SIGMOD*. ACM, 2205–2217.

[47] Haoqiong Bian, Tiannan Sha, and Anastasia Ailamaki. 2023. Using Cloud Functions as Accelerator for Elastic Data Analytics. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 161:1–161:27.

[48] Thomas Bodner. 2020. Elastic Query Processing on Function as a Service Platforms. In *VLDB PhD Workshop*.

[49] Thomas Bodner, Tobias Pietz, Lars Jonas Bollmeier, and Daniel Ritter. 2022. Doppler: Understanding Serverless Query Execution. In *SIGMOD BiDEDE*. ACM, 2:1–2:4.

[50] Marc Brooker. 2019. Timeouts, Retries, and Backoff with Jitter. https://aws.amazon.com/builders-library/timeouts-retries-and-backoff-with-jitter/. Accessed: 2024-10-07.

[51] Marc Brooker. 2023. Surprising Scalability of Multitenancy. https://brooker.co.za/blog/2023/03/23/economics.html. Accessed: 2024-10-07.

[52] Rodrigo Bruno, Serhii Ivanenko, Sutao Wang, Jovan Stevanovic, and Vojin Jovanovic. 2022. Graalvisor: Virtualized Polyglot Runtime for Serverless Applications. *CoRR* abs/2212.10131 (2022). https://doi.org/10.48550/ARXIV.2212.10131

[53] Mengchu Cai, Martin Grund, Anurag Gupta, Fabian Nagel, Ippokratis Pandis, Yannis Papakonstantinou, and Michalis Petropoulos. 2018. Integrated Querying of SQL Database Data and S3 Data in Amazon Redshift. *IEEE Data Engineering Bulletin* 41, 2 (2018), 82–90.

[54] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Engineering Bulletin* 38, 4 (2015), 28–38.

[55] João Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy H. Katz. 2019. Cirrus: A Serverless Framework for End-to-end ML Workflows. In *SoCC*. ACM, 13–24.

[56] Cloud Native Computing Foundation. 2022. 2022 Annual Survey. https://www.cncf.io/reports/cncf-annual-survey-2022/. Accessed: 2024-10-07.

[57] Ben Congdon. 2023. Corral: A Serverless MapReduce Framework Written for AWS Lambda. https://github.com/bcongdon/corral/. Accessed: 2024-10-07.

[58] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. In *Middleware*. ACM, 64–78.

[59] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD*. ACM, 215–226.

[60] Datadog. 2023. The State of Serverless 2023. https://www.datadoghq.com/state-of-serverless/. Accessed: 2024-10-07.

[61] Dominik Durner, Viktor Leis, and Thomas Neumann. 2023. Exploiting Cloud Object Storage for High-Performance Analytics. *PVLDB* 16, 11 (2023), 2769–2782.

[62] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup.

2022. The State of Serverless Applications: Collection, Characterization, and Community Consensus. *IEEE Transactions on Software Engineering* 48, 10 (2022), 4152–4166.

[63] Mostafa Elhemali, Niall Gallagher, Nick Gordon, Joseph Idziorek, Richard Krog, Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somasundaram Perianayagam, Tim Rath, Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, and Akshat Vig. 2022. Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service. In *ATC*. USENIX, 1037–1048.

[64] Google. 2024. Cloud Firestore. https://cloud.google.com/firestore/. Accessed: 2024-10-07.

[65] Google. 2024. Cloud Storage Request Rate and Access Distribution Guidelines. https://cloud.google.com/storage/docs/request-rate/. Accessed: 2024-10-07.

[66] Google. 2024. Google Cloud Functions. https://cloud.google.com/functions/. Accessed: 2024-10-07.

[67] Google. 2024. Google Cloud Functions Version Comparison. https://cloud.google.com/functions/docs/concepts/version-comparison/. Accessed: 2024-10-07.

[68] Google. 2024. Google Cloud Pricing. https://cloud.google.com/pricing/. Accessed: 2024-10-07.

[69] Google. 2024. Serverless. https://cloud.google.com/serverless. Accessed: 2024-10-07.

[70] Martin Grambow, Tobias Pfandzelter, Luk Burchard, Carsten Schubert, Max Xiaohang Zhao, and David Bermbach. 2021. BeFaaS: An Application-Centric Benchmarking Framework for FaaS Platforms. In *IC2E*. IEEE, 1–8.

[71] Jim Gray and Franco Putzolu. 1987. The 5 Minute Rule for Trading Memory for Disc Accesses and the 10 Byte Rule for Trading Memory for CPU Time. In *SIGMOD*. ACM, 395–398.

[72] iPerf3 contributors. 2024. iPerf - The Ultimate Speed Test Tool for TCP, UDP and SCTP. https://iperf.fr/. Accessed: 2024-10-07.

[73] Jananie Jarachanthan, Li Chen, Fei Xu, and Bo Li. 2022. Astrea: Auto-Serverless Analytics Towards Cost-Efficiency and QoS-Awareness. *IEEE Transactions Parallel Distributed Systems* 33, 12 (2022), 3833–3849.

[74] Jiawei Jiang, Shaoduo Gan, Bo Du, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, Sheng Wang, and Ce Zhang. 2024. A Systematic Evaluation of Machine Learning on Serverless Infrastructure. *VLDB Journal* 33, 2 (2024), 425–449.

[75] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. In *SoCC*. ACM, 445–451.

[76] Kafka contributors. 2024. Apache Kafka. https://kafka.apache.org/. Accessed: 2024-10-07.

[77] Svilen Kanev, Juan Pablo Darago, Kim M. Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David M. Brooks. 2015. Profiling a Warehouse-Scale Computer. In *ISCA*. ACM, 158–169.

[78] Jeongchul Kim and Kyungyong Lee. 2019. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *CLOUD*. IEEE, 502–504.

[79] Youngbin Kim and Jimmy Lin. 2018. Serverless Data Analytics with Flint. In *CLOUD*. IEEE, 451–455.

[80] Kyle Kingsbury. 2024. Consistency Models. https://jepsen.io/consistency. Accessed: 2024-10-07.

[81] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *OSDI*. USENIX, 427–444.

[82] Jonathan Lee, Stas Ilinskiy, and William Ehlhardt. 2020. Alki, Or How We Learned to Stop Worrying and Love Cold Metadata. https://dropbox.tech/infrastructure/alki--or-how-we-learned-to-stop-worrying-and-love-cold-metadata. Accessed: 2024-10-07.

[83] Viktor Leis. 2024. SSDs Have Become Ridiculously Fast, Except in the Cloud. http://databasearchitects.blogspot.com/2024/02/ssds-have-become-ridiculously-fast.html. Accessed: 2024-10-07.

[84] Fabian Mahling, Paul Rößler, Thomas Bodner, and Tilmann Rabl. 2023. BabelMR: A Polyglot Framework for Serverless MapReduce. In *Joint Proceedings of Workshops at the 49th International Conference on Very Large Data Bases (CEUR Workshop Proceedings)*, Vol. 3462.

[85] Pascal Maissen, Pascal Felber, Peter G. Kropf, and Valerio Schiavoni. 2020. FaaSdom: A Benchmark Suite for Serverless Computing. In *DEBS*. ACM, 73–84.

[86] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *PVLDB* 13, 12 (2020), 3461–3472.

[87] Microsoft. 2024. Azure Blob Storage: Massively Scalable and Secure Object Storage for Cloud-native Workloads, Archives, Data Lakes, High-performance Computing, and Machine Learning. https://azure.microsoft.com/products/storage/blobs/. Accessed: 2024-10-07.

[88] Microsoft. 2024. Azure Files: Simple, Secure, and Serverless Enterprise-grade Cloud File Shares. https://azure.microsoft.com/products/storage/files/. Accessed: 2024-10-07.

[89] Microsoft. 2024. Azure Functions: Execute Event-driven Serverless Code with an End-to-end Development Experience. https://azure.microsoft.com/services/functions/. Accessed: 2024-10-07.

[90] Microsoft. 2024. Azure Functions Hosting Options. https://learn.microsoft.com/en-us/azure/azure-functions/functions-scale/. Accessed: 2024-10-07.

[91] Microsoft. 2024. Azure Pricing. https://azure.microsoft.com/pricing/. Accessed: 2024-10-07.

[92] Microsoft. 2024. Azure Serverless: Go Serverless — Build Apps Faster without Managing Infrastructure. https://azure.microsoft.com/solutions/serverless/. Accessed: 2024-10-07.

[93] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *SIGMOD*. ACM, 115–130.

[94] ORC contributors. 2024. Apache ORC. https://orc.apache.org/. Accessed: 2024-10-07.

[95] Surya Chaitanya Palepu, Dheeraj Chahal, Manju Ramesh, and Rekha Singhal. 2022. Benchmarking the Data Layer Across Serverless Platforms. In *HPDC HiPS*. ACM, 3–7.

[96] Parquet contributors. 2024. Apache Parquet. https://parquet.apache.org/. Accessed: 2024-10-07.

[97] Matthew Perron, Raul Castro Fernandez, David J. DeWitt, Michael J. Cafarella, and Samuel Madden. 2023. Cackle: Analytical Workload Cost and Performance Stability With Elastic Pools. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 233:1–233:25.

[98] Matthew Perron, Raul Castro Fernandez, David J. DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *SIGMOD*. ACM, 131–141.

[99] Daniel Barcelona Pons and Pedro García López. 2021. Benchmarking Parallelism in FaaS Platforms. *Future Generation Computer Systems* 124 (2021), 268–284.

[100] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *NSDI*. USENIX, 193–206.

[101] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2021. Characterizing and Mitigating the I/O Scalability Challenges for Serverless Applications. In *IISWC*. IEEE, 74–86.

[102] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. 2010. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *PVLDB* 3, 1 (2010), 460–471.

[103] Joel Scheuner and Philipp Leitner. 2020. Function-as-a-Service Performance Evaluation - A Multivocal Literature Review. *Journal of Systems and Software* 170 (2020), 110708.

[104] Korakit Seemakhupt, Brent E. Stephens, Samira Manabi Khan, Sihang Liu, Hassan M. G. Wassel, Soheil Hassas Yeganeh, Alex C. Snoeren, Arvind Krishnamurthy, David E. Culler, and Henry M. Levy. 2023. A Cloud-Scale Characterization of Remote Procedure Calls. In *SOSP*. ACM, 498–514.

[105] Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *ATC*. USENIX, 205–218.

[106] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. *PVLDB* 13, 11 (2020), 2438–2452.

[107] Junjay Tan, Thanaa M. Ghanem, Matthew Perron, Xiangyao Yu, Michael Stonebraker, David J. DeWitt, Marco Serafini, Ashraf Aboulnaga, and Tim Kraska. 2019. Choosing A Cloud DBMS: Architectures and Tradeoffs. *PVLDB* 12, 12 (2019), 2170–2182.

[108] Transaction Processing Performance Council. 2024. Specification of the TPC-H Benchmark. https://www.tpc.org/tpch/. Accessed: 2024-10-07.

[109] Transaction Processing Performance Council. 2024. Specification of the TPCx-BB Benchmark. https://www.tpc.org/tpcx-bb/. Accessed: 2024-10-07.

[110] Dmitrii Ustiugov, Theodor Amariucai, and Boris Grot. 2021. Analyzing Tail Latency in Serverless Clouds with STeLLAR. In *IISWC*. IEEE, 51–62.

[111] Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan S. Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. 2020. Is Big Data Performance Reproducible in Modern Cloud Networks?. In *NSDI*. USENIX, 513–527.

[112] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *NSDI*. USENIX, 449–462.

[113] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael M. Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *ATC*. USENIX, 133–146.

[114] Michal Wawrzoniak, Ingo Müller, Gustavo Alonso, and Rodrigo Bruno. 2021. Boxer: Data Analytics on Network-enabled Serverless Platforms. In *CIDR*. www.cidrdb.org.

[115] WikiChip. 2024. AWS Graviton2. https://en.wikichip.org/wiki/annapurna_labs/graviton/graviton2. Accessed: 2024-10-07.

[116] Andreas Wittig. 2023. Worldwide Availability of EC2 Instance Types. https://cloudonaut.io/worldwide-availability-of-ec2-instance-types/. Accessed: 2024-10-07.

[117] David Yanacek. 2020. Fairness in Multi-tenant Systems. https://aws.amazon.com/builders-library/fairness-in-multi-tenant-systems/. Accessed: 2024-10-07.

[118] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing Serverless Platforms with ServerlessBench. In *SoCC*. ACM, 30–44.