Pocket: Elastic Ephemeral Storage for Serverless Analytics

Ana Klimovic and Yawen Wang, ***Stanford University;*** **Patrick Stuedi, Animesh Trivedi, and Jonas Pfefferle,** ***IBM Research;*** **Christos Kozyrakis,** ***Stanford University***

https://www.usenix.org/conference/osdi18/presentation/klimovic

This paper is included in the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18).

October 8–10, 2018 • Carlsbad, CA, USA

ISBN 978-1-939133-08-3

Open access to the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

Pocket: Elastic Ephemeral Storage for Serverless Analytics

Ana Klimovic[1] Yawen Wang[1] Patrick Stuedi [2]

Animesh Trivedi[2] Jonas Pfefferle[2] Christos Kozyrakis[1]

[1] Stanford University [2] IBM Research

Abstract

Serverless computing is becoming increasingly popular, enabling users to quickly launch thousands of short-lived tasks in the cloud with high elasticity and fine-grain billing. These properties make serverless computing appealing for interactive data analytics. However exchanging intermediate data between execution stages in an analytics job is a key challenge as direct communication between serverless tasks is difficult. The natural approach is to store such ephemeral data in a remote data store. However, existing storage systems are not designed to meet the demands of serverless applications in terms of elasticity, performance, and cost. We present *Pocket*, an elastic, distributed data store that automatically scales to provide applications with desired performance at low cost. Pocket dynamically rightsizes resources across multiple dimensions (CPU cores, network bandwidth, storage capacity) and leverages multiple storage technologies to minimize cost while ensuring applications are not bottlenecked on I/O. We show that Pocket achieves similar performance to ElastiCache Redis for serverless analytics applications while reducing cost by almost 60%.

1 Introduction

Serverless computing is becoming an increasingly popular cloud service due to its high elasticity and fine-grain billing. Serverless platforms like AWS Lambda, Google Cloud Functions, and Azure Functions enable users to quickly launch thousands of light-weight tasks, as opposed to entire virtual machines. The number of serverless tasks scales automatically based on application demands and users are charged only for the resources their tasks consume, at millisecond granularity [17, 36, 56].

While serverless platforms were originally developed for web microservices and IoT applications, their elasticity and billing advantages make them appealing for data intensive applications such as *interactive analytics*. Several recent frameworks launch large numbers of fine-grain tasks on serverless platforms to exploit all available parallelism in an analytics job and achieve near real-time performance [32, 45, 27]. In contrast to traditional serverless applications that consist of a single function executed when a new request arrives, analytics jobs typically consist of multiple stages and require sharing of state and data across stages of tasks (e.g., data shuffling).

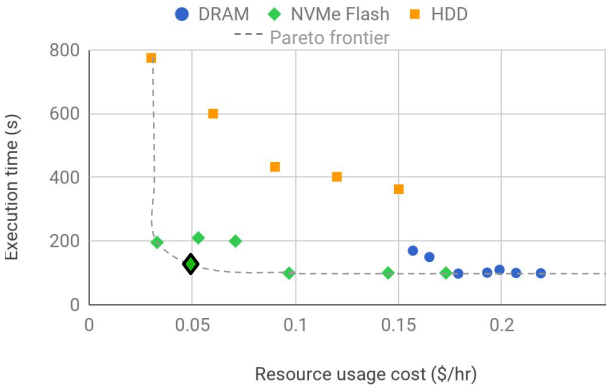

Figure 1: Example of performance-cost trade-off for a serverless video analytics job using different storage technologies and VM types in Amazon EC2

Most analytics frameworks (e.g., Spark) implement data sharing with a long-running framework agent on each node buffering intermediate data in local storage [78]. This enables tasks from different execution stages to directly exchange intermediate data over the network. However, in serverless deployments, there is no long-running application framework agent to manage local storage. Furthermore, serverless applications have no control over task scheduling or placement, making direct communication among tasks difficult. As a result of these limitations, the natural approach for data sharing in serverless applications is to use a remote storage service. For instance, early frameworks for serverless analytics either use object stores (e.g., S3 [16]), databases (e.g., CouchDB [1]) or distributed caches (e.g., Redis [51]).

Unfortunately, existing storage services are not a good fit for sharing short-lived intermediate data in serverless applications. We refer to the intermediate data as

ephemeral data to distinguish it from input and output data which requires long-term storage. File systems, object stores and NoSQL databases prioritize providing durable, long-term, and highly-available storage rather than optimizing for performance and cost. Distributed key-value stores offer good performance, but burden users with managing the storage cluster scale and configuration, which includes selecting the appropriate compute, storage and network resources to provision.

The availability of different storage technologies (e.g., DRAM, NVM, Flash, and HDD) increases the complexity of finding the best cluster configuration for performance and cost. However, the choice of storage technology is critical since jobs may exhibit different storage latency, bandwidth and capacity requirements while different storage technologies vary significantly in terms of their performance characteristics and cost [48]. As an example, Figure 1 plots the performance-cost tradeoff for a serverless video analytics application using a distributed ephemeral data store configured with different storage technologies, number of nodes, compute resources per node, and network bandwidth (see §6.1 for our AWS experiment setup). Each resource configuration leads to different performance and cost. Finding Pareto efficient storage allocations for a job is non-trivial and gets more complicated with multiple jobs.

We present *Pocket*, a distributed data store designed for efficient data sharing in serverless analytics. Pocket offers high throughput and low latency for arbitrary size data sets, automatic resource scaling, and intelligent data placement across multiple storage tiers such as DRAM, Flash, and disk. The unique properties of Pocket result from a strict separation of responsibilities across three planes: a control plane which determines data placement policies for jobs, a metadata plane which manages distributed data placement, and a 'dumb' (i.e., metadata-oblivious) data plane responsible for storing data. Pocket scales all three planes independently at fine resource and time granularity based on the current load. Pocket uses heuristics, which take into account job characteristics, to allocate the right storage media, capacity, bandwidth and CPU resources for cost and performance efficiency. The storage API exposes deliberately simple I/O operations for sub-millisecond access latency. We intend for Pocket to be managed by cloud providers and offered to users with a pay-what-you-use cost model.

We deploy Pocket on Amazon EC2 and evaluate the system using using three serverless analytics workloads: video analytics, MapReduce sort, and distributed source code compilation. We show that Pocket is capable of rightsizing the type and number of resources such that jobs achieve similar performance compared to using ElastiCache Redis, a DRAM-based key-value store, while saving almost 60% in cost.

In summary, our contributions are as follows:

- We identify the key characteristics of ephemeral data in serverless analytics and synthesize requirements for storage platforms used to share such data among serverless tasks.
- We introduce Pocket, a distributed data store whose control, metadata and data planes are designed for sub-second response times, automatic resource scaling and intelligent data placement across storage tiers. To our knowledge, Pocket is the first platform targeting data sharing in serverless analytics.
- We show that Pocket's data plane delivers sub-millisecond latency and scalable bandwidth while the control plane rightsizes resources based on the number of jobs and their attributes. For a video analytics job, Pocket reduces the average time serverless tasks spend on ephemeral I/O by up to 4.1× compared to S3 and achieves similar performance to ElastiCache Redis while saving 59% in cost.

Pocket is open-source software. The code is available at: `https://github.com/stanford-mast/pocket`.

2 Storage for Serverless Analytics

Early work in serverless analytics has identified the challenge of storing and exchanging data between hundreds of fine-grain, short-lived tasks [45, 32]. We build on our study of ephemeral storage requirements for serverless analytics applications [49] to synthesize essential properties for an ephemeral data storage solution. We also discuss why current systems are not able to meet the ephemeral I/O demands of serverless analytics applications. Our focus is on ephemeral data as the original input and final output data of analytics jobs typically has long-term availability and durability requirements that are well served by the variety of file systems, object stores, and databases available in the cloud.

2.1 Ephemeral Storage Requirements

High performance for a wide range of object sizes: Serverless analytics applications vary considerably in the way they store, distribute, and process data. This diversity is reflected in the granularity of ephemeral data that is generated during a job. Figure 2 shows the ephemeral object size distribution for a distributed lambda compilation of the `cmake` program, a serverless video analytics job using the Thousand Island (THIS) video scanner [63], and a 100 GB MapReduce sort job on lambdas. The key observation is that ephemeral data access granularity varies greatly in size, ranging from hundreds of bytes to hundreds of megabytes. We observe a

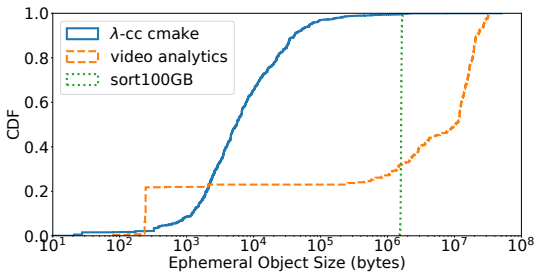

Figure 2: Objects are 100s of bytes to 100s of MBs.

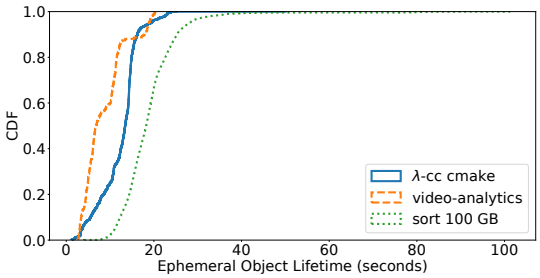

Figure 3: Objects have short lifetime.

straight line for sorting as its ephemeral data size is equal to the partition size. However, with a different dataset size and/or number of workers, the location of the line changes. Applications that read/write large objects demand high throughput (e.g., we find that sorting 100 GB with 500 lambdas requires up to 7.5 GB/s of ephemeral storage throughput) while low latency is important for small object accesses. *Thus, an ephemeral data store must deliver high bandwidth, low latency, and high IOPS for the entire range of object sizes.*

Automatic and fine-grain scaling: One of the key promises of serverless computing is agility to dynamically meet application demands. Serverless frameworks can launch thousands of short-lived tasks instantaneously. Thus, an ephemeral data store for serverless applications can observe a storm of I/O requests within a fraction of a second. Once the load dissipates, the storage (just like the compute) resources should be scaled down for cost efficiency. Scaling up or down to meet elastic application demands requires a storage solution capable of growing and shrinking in multiple resource dimensions (e.g., adding/removing storage capacity and bandwidth, network bandwidth, and CPU cores) at a fine time granularity on the order of seconds. In addition, users of serverless platforms desire a storage service that automatically manages resources and charges users only for the fine-grain resources their jobs actually consume, so as to match the abstraction that serverless computing already provides for compute and memory resources. Automatic resource management is important since navigating cluster configuration performance-cost trade-offs is a burden for users. For example, finding the Pareto optimal point outlined in Figure 1 is non-trivial; it is the point beyond which adding resources only increases cost without improving execution time while using any lower-cost resource allocation results in sub-optimal execution time. In summary, *an ephemeral data store must automatically rightsize resources to satisfy application I/O requirements while minimizing cost.*

Storage technology awareness: In addition to rightsizing cluster resources, the storage system also needs to decide which storage technology to use for which data. The variety of storage media available in the cloud allow for different performance-cost trade-offs, as shown in Figure 1. Each storage technology differs in terms of I/O latency, throughput and IOPS per GB of capacity, and the cost per GB. The optimal choice of storage media for a job depends on its characteristics. *Hence, the ephemeral data store must place application data on the right storage technology tier(s) for performance and cost efficiency.*

Fault-(in)tolerance: Typically a data store must deal with failures while keeping the service up and running. Hence, it is common for storage systems to use fault-tolerance techniques such as replication and erasure codes [42, 66, 46]. For data that needs to be stored long-term, such as the original input and final output data for analytics workloads, the cost of data unavailability typically outweighs the cost of fault-tolerance mechanisms. However, as shown in Figure 3, ephemeral data has a short lifetime of 10-100s of seconds. Unlike the original input and final output data, ephemeral data is only valuable during the execution of a job and can easily be regenerated. Furthermore, fault tolerance is typically baked into compute frameworks, such that storing the data and computing it become interchangeable [39]. For example, Spark uses a data abstraction called resilient distributed datasets (RDDs) to mitigate stragglers and track lineage information for fast data recovery [78]. *Hence, we argue that an ephemeral storage solution does not have to provide high fault-tolerance as expected of traditional storage systems.*

2.2 Existing Systems

Existing storage systems do not satisfy the combination of requirements outlined in § 2.1. We describe different categories of systems and summarize why they fall short for elastic ephemeral storage in Table 1.

Severless applications commonly use fully-managed cloud storage services, such as Amazon S3, Google Cloud Storage, and DynamoDB. These systems extend the 'serverless' abstraction to storage, charging users only for the capacity and bandwidth they use [16, 28].

	Elastic scaling	Latency	Throughput	Max object size	Cost
S3	Auto, coarse-grain	High	Medium	5 TB	$
DynamoDB	Auto, fine-grain, pay per hour	Medium	Low	400 KB	$$
Elasticache Redis	Manual	Low	High	512 MB	$$$
Aerospike	Manual	Low	High	1 MB	$$
Apache Crail	Manual	Low	High	any size	$$
Desired for λs	*Auto, fine-grain, pay per second*	*Low*	*High*	*any size*	*$*

Table 1: Comparison of existing storage systems and desired properties for ephemeral storage in serverless analytics.

While such services automatically scale resources based on usage, they are optimized for high durability hence their agility is limited and they do not meet the performance requirements of serverless analytics applications. For example, S3 has high latency overhead (e.g., a 1 KB read takes ∼12 ms) and insufficient throughput for highly parallel applications. For example, sorting 100 GB with 500 or more workers results in request rate limit errors when S3 is used for intermediate data.

In-memory key-value stores, such as Redis and Memcached, provide another option for storing ephemeral data [51, 8]. These systems offer low latency and high throughput but at the higher cost of DRAM. They also require users to manage their own storage instances and manually scale resources. Although Amazon and Azure offer managed Redis clusters through their ElastiCache and Redis Cache services respectively, they do not automate storage management as desired by serverless applications [13, 57]. Users must still select instance types with the appropriate memory, compute and network resources to match their application requirements. In addition, changing instance types or adding/removing nodes can require tearing down and restarting clusters, with nodes taking minutes to start up while the service is billed for hourly usage.

Another category of systems use Flash storage to decrease cost, while still offering good performance. For example, Aerospike is a popular Flash-based NoSQL database [69]. Alluxio/Tachyon is designed to enable fast and fault-tolerant data sharing between multiple jobs [53]. Apache Crail is a distributed storage system that uses multiple media tiers to balance performance and cost [2]. Unfortunately, users must manually configure and scale their storage cluster resources to adapt to elastic job I/O requirements. Finding Pareto optimal deployments for performance and cost efficiency is non-trivial, as illustrated for a single job in Figure 1. Cluster configuration becomes even more complex when taking into account the requirements of multiple overlapping jobs.

3 Pocket Design

We introduce *Pocket*, an elastic distributed storage service for ephemeral data that automatically and dynamically rightsizes storage cluster resource allocations to provide high I/O performance while minimizing cost. Pocket addresses the requirements outlined in §2.1 by applying the following key design principles:

1. **Separation of responsibilities:** Pocket divides responsibilities across three different planes: the control plane, the metadata plane, and the data plane. The control plane manages cluster sizing and data placement. The metadata plane tracks the data stored across nodes in the data plane. The three planes can be scaled independently based on variations in load, as described in §4.2.

2. **Sub-second response time:** All I/O operations are deliberately simple, targeting sub-millisecond latencies. Pocket's storage servers are optimized for fast I/O and are only responsible for storing data (not metadata), making them simple to scale up or down. The controller scales resources at second granularity and balances load by intelligently steering incoming job data. This makes Pocket elastic.

3. **Multi-tier storage:** Pocket leverages different storage media (DRAM, Flash, disk) to store a job's data in the tier(s) that satisfy the I/O demands of the application while minimizing cost (see §4.1).

3.1 System Architecture

Figure 4 shows Pocket's system architecture. The system consists of a logically centralized controller, one or more metadata servers, and multiple data plane storage servers.

The controller, which we describe in §4, allocates storage resources for jobs and dynamically scales Pocket metadata and storage nodes up and down as the number

of jobs and their requirements vary over time. The controller also makes data placement decisions for jobs (i.e., which nodes and storage media to use for a job's data).

Metadata servers enforce coarse-grain data placement policies generated by the controller by steering client requests to appropriate storage servers. Pocket's metadata plane manages data at the granularity of *blocks*, whose size is configurable. We use a 64 KB block size in our deployment. Objects larger than the block size are divided into blocks and distributed across storage servers, enabling Pocket to support arbitrary object sizes. Clients access data blocks on metadata-oblivious, performance-optimized storage servers equipped with different storage media (DRAM, Flash, and/or HDD).

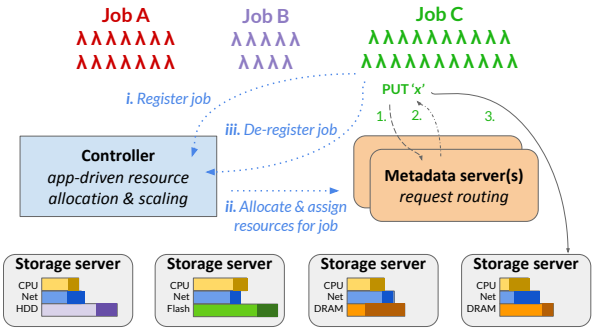

Figure 4: Pocket system architecture and the steps to register job C, issue a PUT from a lambda and de-register the job. The colored bars on storage servers show used and allocated resources for all jobs in the cluster.

3.2 Application Interface

Table 2 outlines Pocket's application interface. Pocket exposes an object store API with additional functions tailored to the ephemeral storage use-case. We describe these functions and how they map to Pocket's separate control, metadata and data planes.

Control functions: Applications use two API calls, `register_job` and `deregister_job`, to interact with the Pocket controller. The `register_job` call accepts hints about a job's characteristics (e.g., degree of parallelism, latency-sensitivity) and requirements (e.g., capacity, throughput). These optional hints help the controller rightsize resource allocations to optimize performance and cost (see §4.1). The `register_job` call returns a job identifier and the metadata server(s) assigned for managing the job's data. The `deregister_job` call notifies the controller that a serverless job has completed.

Metadata functions: While control API calls are issued once per job, serverless tasks in a job can interact with Pocket metadata servers multiple times during their lifetime to write and read ephemeral data. Serverless clients use the `connect` call to establish a connection with Pocket's metadata service. Data in Pocket is stored in objects which are organized in buckets. Objects and buckets are identified using names (strings). Clients can create and delete buckets and enumerate objects in buckets by passing their job identifier and the bucket name. Clients can also lookup and delete existing objects. These metadata operations are similar to those supported by other object stores like Amazon S3.

In our current design, Pocket stores all of a job's data in a top-level bucket identified by the job's ID, which is created during job registration by the controller. This implies each job is assigned to a single metadata server, since a bucket is only managed by one metadata server, to simplify consistency management. However, a job is not fundamentally limited to one metadata server. In general, jobs can create multiple top-level buckets which hash to different metadata servers. In §6.2 we show that a single metadata server in our deployment supports 175K requests per second, which for the applications we study is sufficient to support jobs with thousands of lambdas.

Storage functions: Clients `put` and `get` data to/from objects at a byte granularity. Clients provide their job identifier for all operations. `Put` and `get` operations first involve a metadata lookup. Pocket enhances the basic `put` and `get` object store API calls by accepting an optional data lifetime management hint for these two calls. Since ephemeral data is usually only valuable during the execution of a job, Pockets default coarse-grained behavior is to delete a job's data when the job deregisters. However, applications can set flags to override the default deletion policy for particular objects.

If a client issues a `put` with the PERSIST flag set to true, the object will persist after the job completes. The object is stored on long-running Pocket storage nodes (see §4.2) and will remain in Pocket until it is explicitly deleted or a (configurable) timeout period has elapsed. The ability to persist objects beyond the duration of a job is useful for piping data between jobs. If a client issues a `get` with the DELETE flag set to true, the object will be deleted as soon as it is read, allowing for more efficient garbage collection. Our analysis of ephemeral I/O characteristics for serverless analytics applications reveals that ephemeral data is often written and read only once. For example, a mapper writes an intermediate object destined to a particular reducer. Such data can be deleted as soon as it is consumed instead of waiting for the job to complete and deregister.

3.3 Life of a Pocket Application

We now walk through the life of a serverless analytics application using Pocket. Before launching lambdas, the application first registers with the controller and option-

Client API Function	Description
register_job(jobname, hints=None)	register job with controller and provide optional hints, returns a job ID and metadata server IP address
deregister_job(jobid)	notify controller job has finished, delete job's non-PERSIST data
connect(metadata_server_address)	open connection to metadata server
close()	close connection to metadata server
create_bucket(jobid, bucketname)	create a bucket
delete_bucket(jobid, bucketname)	delete a bucket
enumerate(jobid, bucketname)	enumerate objects in a bucket
lookup(jobid, obj_name)	return `true` if obj_name data exists, else `false`
delete(jobid, obj_name)	delete data
put(jobid, src_filename, obj_name, PERSIST=`false`)	write data, set PERSIST flag if want data to remain after job finishes
get(jobid, dst_filename, obj_name, DELETE=`false`)	read data, set DELETE `true` if data is only read once

Table 2: Main control, metadata, and storage functions exposed by Pocket's client API.

ally provides hints about the job's characteristics (step i in Figure 4). The controller determines the storage tier to use (DRAM, Flash, disk) and the number of storage servers across which to distribute the job's data to meet its throughput and capacity requirements. The controller generates a weight map, described in §4.1, to specify the job's data placement policy and sends this information to the metadata server which it assigned for managing the job's metadata and steering client I/O requests (step ii). If the controller needs to launch new storage servers to satisfy a job's resource allocation, the job registration call stalls until these nodes are available.

When registration is complete, the job launches lambdas. Lambdas first connect to their assigned metadata server, whose IP address is provided by the controller upon job registration. Lambda clients write data by first contacting the metadata server to get the IP address and block address of the storage server to write data to. For writes to large objects which span multiple blocks, the client requests capacity allocation from the metadata server in a streaming fashion; when the capacity of a single block is exhausted, the client issues a new capacity allocation request to the metadata server. Pocket's client library internally overlaps metadata RPCs for the next block while writing data for the current block to avoid stalls. Similarly, lambdas read data by first contacting the metadata server in a similar fashion. Clients cache metadata in case they need to read an object multiple times.

When the last lambda in a job finishes, the job deregisters the job to free up Pocket resources (step iii). Meanwhile, as jobs execute, the controller ontinuously monitors resource utilization in storage and metadata servers (the horizontal bars on storage servers in Figure 4) to add/remove servers as needed to minimize cost while providing high performance (see §4.2).

3.4 Handling Node Failures

Though Pocket is not designed to provide high data durability, the system has mechanisms in place to deal with node failures. Storage servers send heartbeats to the controller and metadata servers. When a storage server fails to send heartbeats, metadata servers automatically mark its blocks as invalid. As a result, client read operations to data that was stored on the faulty storage server will return a 'data unavailable' error. Pocket currently expects the application framework to re-launch serverless tasks to regenerate lost ephemeral data. A common approach is for application frameworks to track data lineage, which is the sequence of tasks that produces each object [78, 39]. For metadata fault tolerance, Pocket supports logging of all metadata RPC operations on shared storage. When a metadata server fails, its state can be reconstructed by replaying the shared log. Controller fault tolerance can be achieved through master-slave replication, though we do not evaluate this in our study.

4 Rightsizing Resource Allocations

Pocket's control plane elastically and automatically rightsizes cluster resources. When a job registers, Pocket's controller leverages optional hints passed through the API to conservatively estimate the job's latency, throughput and capacity requirements and find a cost-effective resource assignment, as described in §4.1. In addition to rightsizing resource allocations for jobs upfront, Pocket continuously monitors the cluster's overall utilization and decides when and how to scale storage and metadata nodes based on load. §4.2 describes Pocket's resource scaling mechanisms along with its data steering policy to balance load.

Hint	Impact on throughput T	Impact on capacity C	Impact on storage media
No hint (default policy)	$T = T_{\text{default}}$ ($T = 50\times$ 8 Gb/s)	$C = C_{\text{default}}$ ($C = 50\times$ 1960 GB)	Fill storage tiers in order of high to low performance (DRAM first, then Flash)
Latency sensitivity	-	-	If latency sensitive, use default policy above. Otherwise, choose the storage tier with the lowest cost for the estimated throughput T and capacity C required for the job.
Maximum number of concurrent lambdas N	$T = N\times$ per-λ Gb/s limit ($T = N \times 0.6$ Gb/s)	$C \propto N\times$ per-λ Gb/s limit ($C = \frac{N\times 0.6}{8\text{ Gb/s}} \times$ 1960 GB)	
Total ephemeral data capacity D	$T \propto D$ ($T = \frac{D}{1960\text{GB}} \times 8$ Gb/s)	$C = D$	
Peak aggregate bandwidth B	$T = B$	$C \propto B$ ($C = \frac{B}{8\text{ Gb/s}} \times 1960\text{GB}$)	

Table 3: The impact that hints provided about the application have on Pocket's resource allocation decisions for throughput, capacity and the choice of storage media (with specific examples in parentheses for our AWS deployment with `i3.2xl` instances, each with 8 cores, 60 GB DRAM, 1.9 TB Flash and ∼8 Gb/s network bandwidth).

4.1 Rightsizing Application Allocation

When a job registers, the controller first determines its *resource allocation* across three dimensions: throughput, capacity, and the choice of storage media. The controller then uses an online bin-packing algorithm to translate the resource allocation into a *resource assignment* on nodes.

Determining job I/O requirements: Pocket uses heuristics that adapt to optional hints passed through the `register_job` API. Table 3 lists the hints that Pocket supports and their impact on the throughput, capacity, and choice of storage media allocated for a job, with examples (in parentheses) for our deployment on AWS.

Given no hints about a job, Pocket uses a default resource allocation that conservatively over-provisions resources to achieve high performance, at high cost. In our AWS deployment, this consists of 50 `i3.2xl` nodes, providing DRAM and NVMe Flash storage with 50 GB/s aggregate throughput. By default, Pocket conservatively assumes that a job is latency sensitive. Hence, Pocket fills the job's DRAM resources before spilling to other storage tiers, in order of increasing storage latency. If a job hints that it is not sensitive to latency, the controller does not allocate DRAM for the job and instead uses the most cost-effective storage technology for the throughput and capacity the controller estimates the job needs.

Knowing a job's maximum number of concurrent lambdas, N, allows Pocket to compute a less conservative estimate of the job's throughput requirement. If this hint is provided, Pocket allocates throughput equal to N times the peak network bandwidth limit per lambda (e.g., ∼600 Mb/s per lambda on AWS). N can be limited by the job's inherent parallelism or the cloud provider's task invocation limit (e.g., 1000 default on AWS).

Pocket's API also accepts hints for the aggregate throughput and capacity requirements of a job, which override Pocket's heuristic estimates. This information can come from profiling. When Pocket receives a throughput hint with no capacity hint, the controller allocates capacity proportional to the job's throughput allocation. The proportion is set by the storage throughput to capacity ratio on the VMs used (e.g., `i3.2xl` instances in AWS provide 1.9 TB of capacity per ∼ 8 Gb/s of network bandwidth). Vice versa, if only a capacity hint is provided, Pocket allocates throughput based on the VM capacity:throughput ratio. In the future, we plan to allow jobs to specify their average *per-lambda* throughput and capacity requirements, as these can be more meaningful than aggregate throughput and capacity hints for a job when the number of lambdas used is subject to change.

The hints in Table 3 can be specified by application developers or provided by the application framework. For example, the framework we use to run lambda-distributed software compilation automatically infers and synthesizes a job's dependency graph [31]. Hence, this framework can provide Pocket with hints about the job's maximum degree of parallelism, for instance.

Assigning resources: Pocket translates a job's resource allocation into a resource assignment on specific storage servers by generating a *weight map* for the job. The weight map is an associative array mapping each storage server (identified by its IP address and port) to a weight from 0 to 1, which represents the fraction of a job's dataset to place on that storage server. If a storage server is assigned a weight of 1 in a job's weight map, it will store all of the job's data. The controller sends the weight map to metadata servers, which enforce the data placement policy by routing client requests to storage servers using weighted random selection based on the weights in the job's weight map.

The weight map depends on the job's resource requirements and the available cluster resources. Pocket uses an online bin-packing algorithm which first tries to fit a job's throughput, capacity and storage media allocation on active storage servers and only launches new servers if the job's requirements cannot be satisfied by

sharing resources with other jobs [67]. If a job requires more resources than are currently available, the controller launches the necessary storage nodes while the application waits for its job registration command to return. Nodes take a few seconds or minutes to launch, depending on whether a new VM is required (§6.2).

4.2 Rightsizing the Storage Cluster

In addition to rightsizing the storage allocation for each job, Pocket dynamically scales cluster resources to accommodate elastic application load for multiple jobs over time. At its core, the Pocket cluster consists of a few long-running nodes used to run the controller, the minimum number of metadata servers (one in our deployment), and the minimum number of storage servers (two in our deployment). In particular, data written with the PERSIST flag described in §3.2, which has longer lifetime, is always stored on long-running storage servers in the cluster. Beyond these persistent resources, Pocket scales resources on demand based on load. We first describe the mechanism for horizontal and vertical scaling and then discuss the policy Pocket uses to balance cluster load by carefully steering requests across servers.

Mechanisms: The controller monitors cluster resource utilization by processing heartbeats from storage and metadata servers containing their CPU, network, and storage media capacity usage. Nodes send statistics to the controller every second. The interval is configurable.

When launching a new storage server, the controller provides the IP addresses of all metadata servers that the storage server must establish connections with to join the cluster. The new storage server registers a portion of its capacity with each of these metadata servers. Metadata servers independently manage their assigned capacity and do not communicate with each other. Storage servers periodically sends heartbeats to metadata servers.

To remove a storage server, the controller blacklists the storage server by assigning it a zero weight in the weight maps of incoming jobs. This ensures that metadata servers do not steer data from new jobs to this node. The controller instructs a randomly selected metadata server to set a 'kill' flag in the heartbeat responses of the blacklisted storage server. The blacklisted storage server waits until its capacity is entirely freed, as jobs terminate and their ephemeral data are garbage collected. The storage server then terminates and releases its resources.

When the controller launches a new metadata server, the metadata server waits for new storage servers to also be launched and register their capacity. To remove a metadata server, the controller sends a 'kill' RPC to the node. The metadata server waits for all the capacity it manages to be freed, then notifies all connected storage servers to close their connections. When all connections close, the metadata server terminates. Storage servers then register their capacity that was managed by the old metadata server across new metadata servers.

In addition to horizontal scaling, the controller manages vertical scaling. When the controller observes that CPU utilization is high and additional cores are available on a node, the controller instructs the node via a heartbeat response to use additional CPU cores.

Cluster sizing policy: Pocket elastically scales the cluster using a policy that aims to maintain overall utilization for each resource type (CPU, network bandwidth, and the capacity of each storage tier) within a target range. The target utilization range can be configured separately for each resource type and managed separately for metadata servers, long-running storage servers (which store data written with the PERSIST flag set) and regular storage servers. For our deployment, we use a lower utilization threshold of 60% and a upper utilization threshold of 80% for all resource dimensions, for both the metadata and storage nodes. The range is empirically tuned and depends on the time it takes to add/remove nodes. Pocket's controller scales down the cluster by removing a storage server if overall CPU, network bandwidth *and* capacity utilization is below the lower limit of the target range. In this case, Pocket removes a storage server belonging to the tier with lowest capacity utilization. Pocket adds a storage server if overall CPU, network bandwidth *or* capacity utilization is above the upper limit of the target range. To respond to CPU load spikes or lulls, Pocket first tries to vertically scale CPU resources on metadata and storage servers before horizontally scaling the number of nodes.

Balancing load with data steering: To balance load while dynamically sizing the cluster, Pocket leverages the short-lived nature of ephemeral data and serverless jobs. As ephemeral data objects only live for tens to hundreds of seconds (see Figure 3), migrating this data to re-distribute load when nodes are added or removed has high overhead. Instead, Pocket focuses on steering data for incoming jobs across active and new storage servers joining the cluster. Pocket controls data steering by assigning specific weights for storage servers in each job's weight map. To balance load, the controller assigns higher weights to under-utilized storage servers.

The controller uses a similar approach, at a coarser granularity, to balance load across metadata servers. As noted in §3.2, the controller currently assigns each job to one metadata server. The controller estimates the load a job will impose on a metadata server based on its throughput and capacity allocation. Combining this estimate with metadata server resource utilization statistics, the controller selects a metadata server to use for an incoming job such that the predicted metadata server resource utilization remains within the target range.

5 Implementation

Controller: Pocket's controller, implemented in Python, leverages the Kubernetes container orchestration system to launch and tear down metadata and storage servers, running in separate Docker containers [7]. The controller uses Kubernetes Operations (kops) to spin up and down virtual machines that run containers [6]. As explained in §4.2, Pocket rightsizes cluster resources to maintain a target utilization range. We implement a resource monitoring daemon in Python which runs on each node, sending CPU and network utilization statistics to the controller every second. Metadata servers also send storage tier capacity utilization statistics. We empirically tune the target utilization range based on node startup time. For example, we use a conservative target utilization range when the controller needs to launch new VMs compared to when VMs are running and the controller simply launches containers.

Metadata management: We implement Pocket's metadata and storage server architecture on top of the Apache Crail distributed data store [2, 71]. Crail is designed for low latency, high throughput storage of arbitrarily sized data with low durability requirements. Crail provides a unified namespace across a set of heterogeneous storage resources distributed in a cluster. Its modular architecture separates the data and metadata plane and supports pluggable storage tier and RPC library implementations. While Crail is originally designed for RDMA networks, we implement a TCP-based RPC library for Pocket since RDMA is not readily available in public clouds. Like Crail, Pocket's metadata servers are implemented in Java. Each metadata server logs its metadata operations to a file on a shared NFS mount point, such that the log can be accessed and replayed by a new metadata server in case a metadata server fails.

Storage tiers: We implement three different storage tiers for Pocket. The first is a DRAM tier implemented in Java, using NIO APIs to efficiently serve requests from clients over TCP connections. The second tier uses NVMe Flash storage. We implement Pocket's NVMe storage servers on top of ReFlex, a system that allows clients (i.e., lambdas) to access Flash over commodity Ethernet networks with high performance [47]. ReFlex is implemented in C and leverages Intel's DPDK [43] and SPDK [44] libraries to directly access network and NVMe device queues from userspace. ReFlex uses a polling-based execution model to efficiently process network storage requests over TCP. The system also uses a quality of service (QoS) aware scheduler to manage read/write interference on Flash and provide predictable performance to clients. The third tier we implement is a generic block storage tier that allows Pocket to use any block storage device (e.g., HDD or SATA/SAS SSD) via a standard kernel device driver. Similar to ReFlex, this tier is implemented in C and uses DPDK for efficient, userspace networking. However, instead of using SPDK to access NVMe Flash devices from userspace, this tier uses the Linux libaio library to submit asynchronous block storage requests to a kernel block device driver. Leveraging userspace APIs for the Pocket NVMe and generic block device tiers allows us to increase performance and resource efficiency. For example, ReFlex can process up to 11× more requests per core than a conventional Linux network-storage stack [47].

Pocket server	EC2 server	DRAM (GB)	Storage (TB)	Network (Gb/s)	$ / hr
Controller	`m5.xl`	16	0	∼8	0.192
Metadata	`m5.xl`	16	0	∼8	0.192
DRAM	`r4.2xl`	61	0	∼8	0.532
NVMe	`i3.2xl`	61	1.9	∼8	0.624
SSD	`i2.2xl`	61	1.6	$\lesssim 2$	1.705[1]
HDD	`h1.2xl`	32	2	∼8	0.468

Table 4: Type and cost of EC2 VMs used for Pocket

Client library: Since the serverless applications we use are written in Python, we implement Pocket's application interface (Table 2) as a Python client library. The core of the library is implemented in C++ to optimize performance. We use `Boost` to wrap the code into a Python library. The library internally manages TCP connections with metadata and storage servers.

6 Evaluation

6.1 Methodology

We deploy Pocket on Amazon Web Service (AWS). We use EC2 instances to run Pocket storage, metadata, and controller nodes. We use four different kinds of storage media: DRAM, NVMe-based Flash, SATA/SAS-based Flash (which we refer to as SSD), and HDD. DRAM servers run on `r4.2xl` instances, NVMe Flash servers run on `i3.2xl` instances, SSD servers run on `i2.2xl` instances, and HDD servers run on `h1.2xl` instances. We choose the instance families based on their local storage media, shown in Table 4. We choose the VM size to provide a good balance of network bandwidth and storage capacity for the serverless applications we study.

We run Pocket storage and metadata servers as containers on EC2 VMs, orchestrated with Kubernetes v1.9. We use AWS Lambda as our serverless computing platform. We enable lambdas to access Pocket EC2 nodes by deploying them in the same virtual private cloud (VPC).

[1] The cost of the `i2` instance is particularly high since it is an old generation instance that is being phased out by AWS and replaced by the newer generation `i3` instances with NVMe Flash devices.

We configure lambdas with 3 GB of memory. Amazon allocates lambda compute resources proportional to memory resources [18]. We compare Pocket's performance and cost-efficiency to ElastiCache Redis (cluster-mode enabled) and Amazon S3 [13, 51, 16]. We present results from experiments conducted in April 2018.

We study three different serverless analytics applications, described below. The applications differ in their degree of parallelism, ephemeral object size distribution (Figure 2), and throughput requirements.

Video analytics: We use the Thousand Island Scanner (THIS) for distributed video processing [63]. Lambdas in the first stage read compressed video frame batches, decode, and write the decoded frames to ephemeral storage. Each lambda fetches a 250 MB decoder executable from S3 as it does not fit in the AWS Lambda deployment package. Each first stage lambda then launches second stage lambdas, which read decoded frames from ephemeral storage, compute a MXNET deep learning classification algorithm and output an object detection result. We use a 25 minute video with 40K 1080p frames. We tune the batch size for each stage to minimize the job's end-to-end execution time; the first stage consists of 160 lambdas while the second has 305 lambdas.

MapReduce Sort: We implement a MapReduce sort application on AWS Lambda, similar to PyWren [45]. Map lambdas fetch input files from long-term storage (we use S3) and write intermediate files to ephemeral storage. Reduce lambdas merge and sort intermediate data and upload output files to long-term storage. We run a 100 GB sort, which generates 100 GB of ephemeral data. We run the job with 250, 500, and 1000 lambdas.

Distributed software compilation (λ-cc): We use gg to infer software build dependency trees and invoke lambdas to compile source code with high parallelism [4, 31]. Each lambda fetches its dependencies from ephemeral storage, computes (i.e., compiles, archives or links), and writes its output to ephemeral storage, including the final executable for the user to download. We present results for compiling the `cmake` project source code. This build job has a maximum inherent parallelism of 650 tasks and generates a total of 850 MB ephemeral data. Object size ranges from 10s of bytes to MBs, as shown in Figure 2.

6.2 Microbenchmarks

Storage request latency: Figure 5 compares the 1 KB request latency of S3, Redis, and various Pocket storage tiers measured from a lambda client. Pocket-DRAM, Pocket-NVMe and Redis latency is below 540 μs, which is over 45$\times$ faster than S3. The latency of the Pocket-SSD and Pocket-HDD tiers is higher due to higher storage media access times. Pocket-HDD `get` latency is higher than `put` latency since lambdas issue random reads while writes are sequential; the metadata server routes writes to sequential logical block addresses. Pocket-DRAM has higher latency than Redis mainly due to the metadata lookup RPC, which takes 140 μs. While Redis cluster clients simply hash keys to Redis nodes, Pocket clients must contact a metadata server. While this extra RPC increases request latency, it allows Pocket to optimize data placement per job and dynamically scale the cluster without redistributing data across nodes.

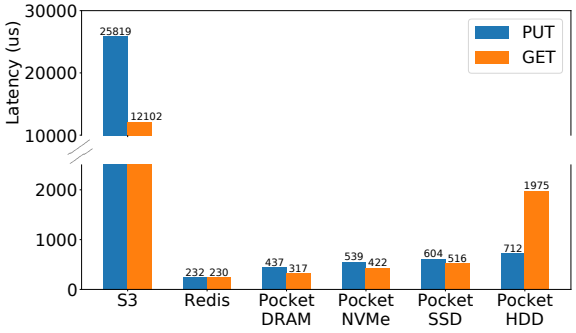

Figure 5: Unloaded latency for 1KB access from lambda

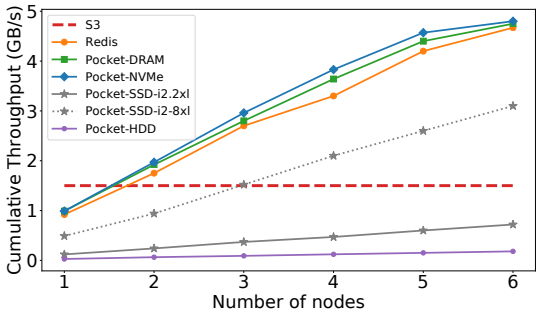

Figure 6: Total GB/s for 1MB requests from 100 lambdas

Storage request throughput: We measure the `get` throughput of S3, Redis (`cache.r4.2xl`) and various Pocket storage tiers by issuing 1 MB requests from 100 concurrent lambdas. In Figure 6, we sweep the number of nodes in the Redis and Pocket clusters and compare the cumulative throughput to that achieved with S3. Pocket-DRAM, Pocket-NVMe and Redis all achieve similar throughput. With a single node, the bottleneck is the 1 GB/s VM network bandwidth. With two nodes, Pocket's DRAM and NVMe tiers achieve higher throughput than S3. Pocket's SSD and HDD tiers have significantly lower throughput. The HDD tier is limited by the 40 MB/s random access bandwidth of the disk on each node. The SSD tier is limited by poor networking (less than $\sim$2 Gb/s) on the old generation `i2.2xl` instances. Hence, we also plot the throughput using `i2.8xl` instances which have 10 Gb/s networking. The

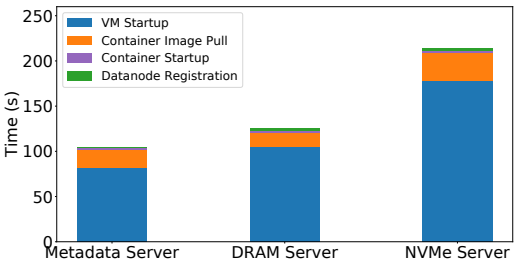

Figure 7: Node startup time breakdown

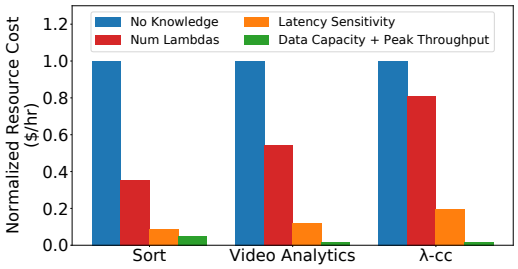

Figure 8: Pocket leverages cumulative hints about job characteristics to allocate resources cost-efficiently.

bottleneck becomes the 500 MB/s throughput limit of the SATA/SAS SSD.

We focus the rest of our evaluation of Pocket on the DRAM and NVMe Flash tiers as they demand the highest data plane software efficiency due to the technology's low latency and high throughput. We also find that in our AWS deployment, the DRAM and NVMe tiers offer significantly higher performance-cost efficiency compared to the HDD and SSD tiers. For example, NVMe Flash servers, which run on on `i3.2xl` instances, provide 1 GB/s per 1900 GB capacity at a cost of $0.624/hour. Meanwhile, HDD servers, which run on `h1.2xl` instances, provide only 40 MB/s per 2000 GB capacity at a cost of $0.468/hour. Thus, the NVMe tier offers 19.7× higher throughput per GB per dollar.

Metadata throughput: We measure the number of metadata operations that a metadata server can handle per second. A single core metadata server on the `m5.xl` instance supports up to 90K operations per second and up to 175K operations per second with four cores. The peak metadata request rate we observe for the serverless analytics applications we study is 75 operations per second per lambda. Hence, a multi-core metadata server can support jobs with thousands of lambdas.

Adding/removing servers: Since Pocket runs in containers on EC2 nodes, we measure the time it takes to launch a VM, pull the container image, and launch the container. Pocket storage servers must also register their storage capacity with metadata servers to join the cluster. Figure 7 shows the time breakdown. VM launch time varies across EC2 instance types. The container image for the metadata server and DRAM server has a compressed size of 249 MB while the Pocket-NVMe compressed container image is 540 MB due to dependencies for DPDK and SPDK to run ReFlex. The image pull time depends on the VM's network bandwidth. The VM launch time and container image pull time only need to be done once when the VM is first started. Once the VM is warm, meaning the image is available locally, starting and stopping containers takes only a few seconds. The time to terminate a VM is tens of seconds.

6.3 Rightsizing Resource Allocations

We now evaluate Pocket with the three different serverless applications described in §6.1.

Rightsizing with application hints: Figure 8 shows how Pocket leverages user hints to make cost-effective resource allocations, assuming each hint is provided in addition to the previous ones. With no knowledge of application requirements, Pocket defaults to a policy that spreads data for a job across a default allocation of 50 nodes, filling DRAM first, then Flash. With knowledge of the maximum number of concurrent lambdas (250, 160, and 650 for the sort, video analytics and λ-cc jobs, respectively), Pocket allocates lower aggregate throughput than the default allocation while maintaining similar job execution time (within 4% of the execution time achieved with the default allocation). Furthermore, these jobs are not sensitive to latency; the sort job and the first stage of the video analytics job are throughput intensive while λ-cc and the second stage of the video analytics job are compute limited. The orange bars in Figure 8 show the cost savings of using NVMe Flash as opposed to DRAM when the latency insensitivity hint is provided for these jobs. The green bar shows the relative resource allocation cost when applications provide explicit hints for their capacity and peak throughput requirements; such hints can be obtained from a profiling run. Across all scenarios, each job's execution time remains within 4% of its execution time with the default resource allocation.

Reclaiming capacity using hints: Figure 9 shows the capacity used over time for the video analytics job, with and without data lifetime management hints. All ephemeral data in this application is written and read only once, since each first stage lambda writes ephemeral data destined to a single second stage lambda. Hence for all `get` operations, this job can make use of the DELETE hint which informs Pocket to promptly garbage collect an object as soon as it has been read. By default, when the DELETE hint is not specified, Pocket waits until the job deregisters to delete the job's data. The job in Figure 9

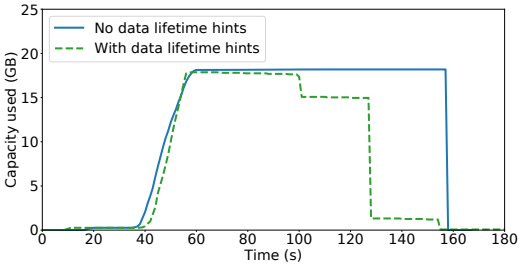

Figure 9: Example of using the DELETE hint for `get` operations in a video analytics job, enabling Pocket to readily reclaim capacity by deleting objects after they have been read versus waiting for the job to complete.

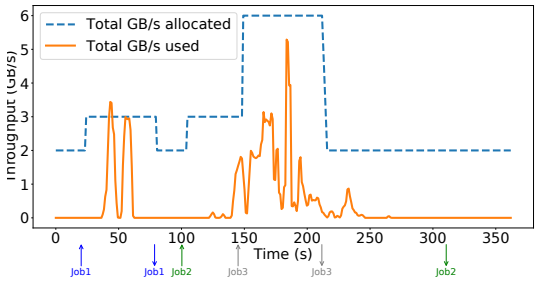

Figure 10: Pocket's controller dynamically scales cluster resources to meet I/O requirements as jobs come and go.

completes at the 158 second mark. We show that leveraging the DELETE hint allows Pocket to reclaim capacity more promptly, making more efficient use of resources as this capacity can be offered to other jobs.

Rightsizing cluster size: Elastic and automatic resource scaling is a key property of Pocket. Figure 10 shows how Pocket scales cluster resources as multiple jobs register and deregister with the controller. Job registration and deregistration times are indicated by upwards and downwards arrows along the x-axis, respectively. In this experiment, we assume Pocket receives capacity and throughput hints for each job's requirements. The first job is a 10 GB sort application requesting 3 GB/s, the second job is a video analytics application requesting 2.5 GB/s and the third job is a different invocation of a 10 GB sort also requesting 3 GB/s. Each storage server provides 1 GB/s. We use a minimum of two storage servers in the cluster. We provision seven VMs for this experiment and ensure that storage server containers are locally available, such that when the controller launches new storage servers, only container startup and capacity registration time is included.

Figure 10 shows that Pocket quickly and effectively scales the allocated storage bandwidth (dotted line) to meet application throughput demands (solid line). The spike surpassing the allocated throughput is due to a short burst in EC2 VM network bandwidth. The VMs provide 'up to 10 Gb/s', but since we typically observe a ∼8 Gb/s bandwidth limit in practice, the controller allocates throughput assuming each node provides 8 Gb/s. As the controller rightsizes resources for each job, job execution time stays within 5% of its execution time when running on 50 nodes, the conservative default resource allocation. If the controller had to spin up new VMs to accommodate a job's requirements instead of just launching containers, the job's start time would be delayed by up to 215 seconds (see EC2 NVMe server startup time in Figure 7) since the `register_job` call blocks until the required storage servers are available.

6.4 Comparison to S3 and Redis

Job execution time: Figure 11 plots the per-lambda execution time breakdown for the MapReduce 100 GB sort job, run with 250, 500, and 1000 concurrent lambdas. The purple bars show the time spent fetching original input data and writing final output data to S3 while the blue bars compare the time for ephemeral data I/O with S3, Redis and Pocket-NVMe. S3 does not support sufficient request rates when the job is run with 500 or more lambdas. S3 returns errors, advising to reduce the I/O rate. Pocket provides similar throughput to Redis, however since the application is not sensitive to latency, Pocket uses NVMe Flash instead of DRAM to reduce cost.

Similarly, for the video analytics job, we observe that Pocket-NVMe achieves the same performance as Redis. However, using S3 for the video analytics job increases the average time spent on ephemeral I/O by each lambda in the first stage (video decoding) by 3.2× and 4.1× for lambdas in the second stage (MXNET classification), compared to using Pocket or Redis.

The performance of the distributed compilation job (λ-cc cmake) is limited by lambda CPU resources [49]. A software build job has inherently limited parallelism; early-stage lambdas compile independent files in parallel, however lambdas responsible for archiving and linking are serialized as they depend on the outputs of the early-stage lambdas. We observe that the early-stage lambdas are compute-bound on current serverless infrastructure. Although using Pocket or Redis reduces the fraction of time each lambda spend on ephemeral I/O, the overall execution time for this job remains the same as when using S3 for ephemeral storage, since the bottleneck is dependencies on compute-bound lambdas.

Cost analysis: Table 4 shows the hourly cost of running Pocket nodes on EC2 VMs in April 2018. Our minimum size Pocket cluster, consisting of one controller node, one metadata server and two `i3.2xl` storage nodes costs $1.632 per hour on EC2. However, Pocket's fixed cost can be amortized as the system is designed to support multiple concurrent jobs from one or more tenants. We intend for Pocket to be operated by a cloud provider

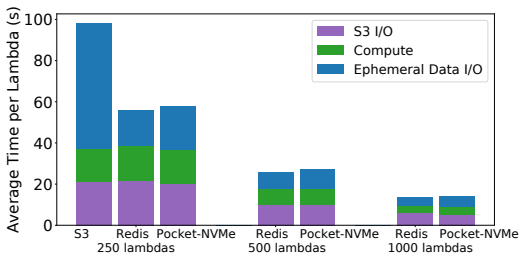

Figure 11: Execution time breakdown of 100GB sort.

Job	S3	Redis	Pocket
100 GB sort	0.05126	5.320	2.1648
Video analytics	0.00034	1.596	0.6483
λ-cc cmake	0.00005	1.596	0.6480

Table 5: Hourly ephemeral storage cost (in USD)

and offered as a storage service with a pay-what-you-use cost model for users, similar to the cost model of serverless computing platforms. Hence, for our cost analysis, we derive fine-grain resource costs, such as the cost of a CPU core and the cost of storage per GB, using AWS EC2 instance pricing. For example, we calculate NVMe Flash $/GB by taking the difference between `i3.2xl` and `r4.2xl` instance costs (since these VMs have the same CPU and DRAM configurations but `i3.2xl` includes a 1900 GB NVMe drive) and dividing by the GB capacity of the `i3.2xl` NVMe drive.

Using this fine-grain resource pricing model for Pocket, Table 5 compares the cost of running the 100 GB sort, video analytics and distributed compilation jobs with S3, ElastiCache Redis, and Pocket-NVMe. We use reduced redundancy pricing for S3 and assume the GB-month cost is charged hourly [15]. We base Redis costs on the price of entire VMs, not only the resources consumed, since ElastiCache Redis clusters are managed by individual users rather than cloud providers. Pocket achieves the same performance as Redis for all three jobs while saving 59% in cost. S3 is still orders of magnitude cheaper. However, S3's cloud provider based cost is not a fair comparison to the cloud user based cost model we use for Pocket and Redis. Furthermore, while the λ-cc job has similar performance with Pocket, Redis and S3 due to a lambda compute bottleneck, the video analytics and sort job execution time is 40 to 65% higher with S3.

7 Discussion

Choice of API: Pocket's simple `get/put` interface provides sufficient functionality for the applications we studied. Lambdas in these jobs consume entire data objects that they read and they do not require updating or appending files. However, POSIX-like I/O semantics for appending or accessing parts of objects could benefit other applications. Pocket's `get/put` API is implemented on top of Apache Crail's append-only stream abstraction which allows clients to read at file offsets and append to files with single-writer semantics [3]. Thus, Pocket's API could easily be modified to expose Crail's I/O semantics. Other operators such as filters or multi-gets could also help optimize the number of RPCs and bytes transferred. The right choice of API for ephemeral storage remains an open question.

Security: Pocket uses access control to secure applications in a multi-tenant environment. To prevent malicious users from accessing other tenants' data, metadata servers issue single-use certificates to clients which are verified at storage servers. An I/O request that is not accompanied with a valid certificate is denied. Clients communicate with metadata servers over SSL to protect against man in the middle attacks. Users set cloud network security rules to prevent TCP traffic snooping on connections between lambdas and storage servers. Alternatively, users can encrypt their data. Pocket does not currently prevent jobs from issuing higher load than specified in job registration hints. Request throttling can be implemented at metadata servers to mitigate interference when a job tries to exceed its allocation.

Learning job characteristics: Pocket currently relies on user or application framework hints to cost-effectively rightsize resource allocations for a job. Currently, Pocket does not autonomously learn application properties. Since users may repeatedly run jobs on different datasets, as many data analytics and modern machine learning jobs are recurring [55], Pocket's controller can maintain statistics about previous invocations of a job and use this information combined with machine learning techniques to rightsize resource allocations for future runs [48, 10]. We plan to explore this in future work.

Applicability to other cloud platforms: While we evaluate Pocket on the AWS cloud platform, the system addresses a real problem applicable across all cloud providers as no available platform provides an optimized way for serverless tasks to exchange ephemeral data. Pocket's performance will vary with network and storage capabilities of different infrastructure. For example, if a low latency network is available, the DRAM storage tier provides significantly lower latency than the NVMe tier. Such variations emphasize the need for a control plane to automate resource allocation and data placement.

Applicability to other cloud workloads: Though we presented Pocket in the context of ephemeral data sharing in serverless analytics, Pocket can also be used for other applications that require distributed, scalable temporary storage. For instance, Google's Cloud Dataflow, a fully-managed data processing service for streaming and batch data analytics pipelines, implements the shuf-

fle operator – used for transforms such as `GroupByKey` – as part of its service backend [35]. Pocket can serve as fast, elastic storage for the intermediate data generated by shuffle operations in this kind of service.

Reducing cost with resource harvesting: Cloud jobs are commonly over-provisioned in terms CPU, DRAM, network, and storage resources due to the difficulty of rigthsizing general jobs and the need to accommodate diurnal load patterns and unexpected load spikes. The result is significant capacity underutilization at the cluster level [21, 74, 29]. Recent work has shown that the plethora of allocated but temporarily unused resources provide a stable substrate that can be used to run analytics job [22, 79]. We can similarly leverage harvested resources to dramatically reduce the total cost of running Pocket. Pocket's storage servers are particularly well suited to run on temporarily idle resource as ephemeral data has short lifetime and low durability requirements.

8 Related Work

Elastic resource scaling: Various reactive [34], predictive [25, 50, 65, 30, 62, 72, 75] and hybrid [24, 41, 33, 60] approaches have been proposed to automatically scale resources based on demand [64, 61]. Muse takes an economic approach, allocating resources to their most efficient use based on a utility function that estimates the impact of resource allocations on job performance [23]. Pocket provisions resources upfront for a job based on hints and conservative heuristics while using a reactive approach to adjust cluster resources over time as jobs enter and leave the system. Pocket's reactive scaling is similar to Horizontal Pod autoscaling in Kubernetes which collects multidimensional metrics and adjusts resources based on utilization ratios [5]. Petal [52] and the controller by Lim et al. [54] propose data re-balancing strategies in elastic storage clusters while Pocket avoids redistributing short-lived data due to the high overhead. CloudScale [68], Elastisizer [40], CherryPick [11], and other systems [73, 77, 48] take an application-centric view to rightsize a job at the coarse granularity of traditional VMs as opposed to determining fine-grain storage requirements. Nevertheless, the proposed cost and performance modeling approaches can also be applied to Pocket to autonomously learn job resource preferences.

Intelligent data placement: Mirador is a dynamic storage service that optimizes data placement for performance, efficiency, and safety [76]. Mirador focuses on long-running jobs (minutes to hours), while Pocket targets short-term (seconds to minutes) ephemeral storage. Tuba manages geo-replicated storage and, similar to Pocket, optimizes data placement based on performance and cost constraints received from applications [20]. Extent-based Dynamic Tiering (EDT) uses access pattern simulations and monitoring to find a cost-efficient storage solution for a workload across multiple storage tiers [38]. The access pattern of ephemeral data is often simple (e.g., write-once-read-once) and the data is short-lived, hence it is not worth migrating between tiers. Multiple systems make storage configuration recommendation based on workload traces [19, 70, 9, 59, 12]. Given I/O traces for a job, Pocket could apply similar techniques to assign resources when a job registers.

Fully managed data warehousing: Cloud providers offer fully managed infrastructure for querying large amounts of structured data with high parallelism and elasticity. Examples include Amazon Redshift [14], Google BigQuery [37], Azure SQL Data Warehouse [58], and Snowflake [26]. These systems are designed to support relational queries and high data durability, while Pocket is designed for elastic, fast, and fully managed storage of data with low durability requirements. However, a cloud data warehouse like Snowflake, which currently stores temporary data generated by query operators on local disk or S3, could leverage Pocket to improve elasticity and resource utilization.

9 Conclusion

General-purpose analytics on serverless infrastructure presents unique opportunities and challenges for performance, elasticity and resource efficiency. We analyzed challenges associated with efficient data sharing and presented Pocket, an ephemeral data store for serverless analytics. In a similar spirit to serverless computing, Pocket aims to provide a highly elastic, cost-effective, and fine-grained storage solution for analytics workloads. Pocket achieves these goals using a strict separation of responsibilities for control, metadata, and data management. To the best of our knowledge, Pocket is the first system designed specifically for ephemeral data sharing in serverless analytics workloads. Our evaluation on AWS demonstrates that Pocket offers high performance data access for arbitrary size data sets, combined with automatic fine-grain scaling, self management and cost effective data placement across multiple storage tiers.

Acknowledgements

We thank our shepherd, Hakim Weatherspoon, and the anonymous OSDI reviewers for their helpful feedback. We thank Qian Li, Francisco Romero, and Sadjad Fouladi for insightful technical discussions. This work is supported by the Stanford Platform Lab, Samsung, and Huawei. Ana Klimovic is supported by a Stanford Graduate Fellowship. Yawen Wang is supported by a Stanford Electrical Engineering Department Fellowship.

References

[1] Apache CouchDB. http://couchdb.apache.org, 2018.

[2] Apache Crail (incubating). http://crail.incubator.apache.org, 2018.

[3] Crail Storage Performance – Part I: DRAM. http://crail.incubator.apache.org/blog/2017/08/crail-memory.html, 2018.

[4] gg: The Stanford Builder. https://github.com/stanfordsnr/gg, 2018.

[5] Horizontal Pod Autoscaler. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale, 2018.

[6] Kubernetes operations (kops). https://github.com/kubernetes/kops, 2018.

[7] Kubernetes: Production-Grade Container Orchestration. https://kubernetes.io, 2018.

[8] Memcached – a distributed memory object caching system. https://memcached.org, 2018.

[9] ALBRECHT, C., MERCHANT, A., STOKELY, M., WALIJI, M., LABELLE, F., COEHLO, N., SHI, X., AND SCHROCK, E. Janus: Optimal flash provisioning for cloud storage workloads. In *Proc. of the USENIX Annual Technical Conference* (2013), ATC'13, pp. 91–102.

[10] ALIPOURFARD, O., LIU, H. H., CHEN, J., VENKATARAMAN, S., YU, M., AND ZHANG, M. CherryPick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), pp. 469–482.

[11] ALIPOURFARD, O., LIU, H. H., CHEN, J., VENKATARAMAN, S., YU, M., AND ZHANG, M. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)* (2017), pp. 469–482.

[12] ALVAREZ, G. A., BOROWSKY, E., GO, S., ROMER, T. H., BECKER-SZENDY, R., GOLDING, R., MERCHANT, A., SPASOJEVIC, M., VEITCH, A., AND WILKES, J. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Trans. Comput. Syst. 19*, 4 (Nov. 2001), 483–518.

[13] AMAZON. Amazon ElastiCache. https://aws.amazon.com/elasticache, 2018.

[14] AMAZON. Amazon redshift. https://aws.amazon.com/redshift, 2018.

[15] AMAZON. Amazon S3 reduced redundancy storage. https://aws.amazon.com/s3/reduced-redundancy, 2018.

[16] AMAZON. Amazon simple storage service. https://aws.amazon.com/s3, 2018.

[17] AMAZON. AWS lambda. https://aws.amazon.com/lambda, 2018.

[18] AMAZON. AWS lambda limits. https://docs.aws.amazon.com/lambda/latest/dg/limits.html, 2018.

[19] ANDERSON, E., HOBBS, M., KEETON, K., SPENCE, S., UYSAL, M., AND VEITCH, A. Hippodrome: Running circles around storage administration. In *Proc. of the 1st USENIX Conference on File and Storage Technologies* (2002), FAST'02, pp. 13–13.

[20] ARDEKANI, M. S., AND TERRY, D. B. A self-configurable geo-replicated cloud storage system. In *Proc. of the 11th USENIX Symposium on Operating Systems Design and Implementation* (2014), OSDI'14, pp. 367–381.

[21] BARROSO, L. A., CLIDARAS, J., AND HLZLE, U. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. 2013.

[22] CARVALHO, M., CIRNE, W., BRASILEIRO, F., AND WILKES, J. Long-term SLOs for reclaimed cloud computing resources. In *Proc. of the ACM Symposium on Cloud Computing* (2014), SOCC '14, pp. 20:1–20:13.

[23] CHASE, J. S., ANDERSON, D. C., THAKAR, P. N., VAHDAT, A. M., AND DOYLE, R. P. Managing energy and server resources in hosting centers. In *Proc. of the Eighteenth ACM Symposium on Operating Systems Principles* (2001), SOSP '01, pp. 103–116.

[24] CHEN, G., HE, W., LIU, J., NATH, S., RIGAS, L., XIAO, L., AND ZHAO, F. Energy-aware server provisioning and load dispatching for connection-intensive internet services. In *Proc. of the 5th USENIX Symposium on Networked Systems Design and Implementation* (2008), NSDI'08, pp. 337–350.

[25] CORTEZ, E., BONDE, A., MUZIO, A., RUSSINOVICH, M., FONTOURA, M., AND BIANCHINI, R. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proc. of the 26th Symposium on Operating Systems Principles* (2017), SOSP '17, pp. 153–167.

[26] DAGEVILLE, B., CRUANES, T., ZUKOWSKI, M., ANTONOV, V., AVANES, A., BOCK, J., CLAYBAUGH, J., ENGOVATOV, D., HENTSCHEL, M., HUANG, J., LEE, A. W., MOTIVALA, A., MUNIR, A. Q., PELLEY, S., POVINEC, P., RAHN, G., TRIANTAFYLLIS, S., AND UNTERBRUNNER, P. The snowflake elastic data warehouse. In *Proc. of the International Conference on Management of Data* (2016), SIGMOD '16, pp. 215–226.

[27] DATABRICKS. Databricks serverless: Next generation resource management for Apache Spark. `https://databricks.com/blog/2017/06/07/databricks-serverless-next-generation-resource-management-for-apache-spark.html`, 2017.

[28] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (2007), SOSP '07, pp. 205–220.

[29] DELIMITROU, C., AND KOZYRAKIS, C. Quasar: Resource-efficient and QoS-aware cluster management. In *Proc. of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (2014), ASPLOS '14, pp. 127–144.

[30] DOYLE, R. P., CHASE, J. S., ASAD, O. M., JIN, W., AND VAHDAT, A. M. Model-based resource provisioning in a web service utility. In *Proc. of the 4th USENIX Symposium on Internet Technologies and Systems* (2003), USITS'03, pp. 5–5.

[31] FOULADI, S., ITER, D., CHATTERJEE, S., KOZYRAKIS, C., ZAHARIA, M., AND WINSTEIN, K. A thunk to remember: make -j1000 (and other jobs) on functions-as-a-service infrastructure (preprint). `http://stanford.edu/~sadjad/gg-paper.pdf`.

[32] FOULADI, S., WAHBY, R. S., SHACKLETT, B., BALASUBRAMANIAM, K. V., ZENG, W., BHALERAO, R., SIVARAMAN, A., PORTER, G., AND WINSTEIN, K. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *Proc. of the 14th USENIX Symposium on Networked Systems Design and Implementation* (2017), NSDI'17, pp. 363–376.

[33] GANDHI, A., CHEN, Y., GMACH, D., ARLITT, M., AND MARWAH, M. Minimizing data center sla violations and power consumption via hybrid resource provisioning. In *Proc. of the 2011 International Green Computing Conference and Workshops* (2011), IGCC '11, pp. 1–8.

[34] GANDHI, A., HARCHOL-BALTER, M., RAGHUNATHAN, R., AND KOZUCH, M. A. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Trans. Comput. Syst. 30*, 4 (Nov. 2012), 14:1–14:26.

[35] GOOGLE. Introducing Cloud Dataflow Shuffle: For up to 5x performance improvement in data analytic pipelines. `https://cloud.google.com/blog/products/gcp/introducing-cloud-dataflow-shuffle-for-up-to-5x-performance-improvement-in-data-analytic-pipelines`, 2017.

[36] GOOGLE. Cloud functions. `https://cloud.google.com/functions`, 2018.

[37] GOOGLE. Google bigquery. `https://cloud.google.com/bigquery`, 2018.

[38] GUERRA, J., PUCHA, H., GLIDER, J., BELLUOMINI, W., AND RANGASWAMI, R. Cost effective storage using extent based dynamic tiering. In *Proc. of the 9th USENIX Conference on File and Stroage Technologies* (2011), FAST'11, pp. 20–20.

[39] GUNDA, P. K., RAVINDRANATH, L., THEKKATH, C. A., YU, Y., AND ZHUANG, L. Nectar: Automatic management of data and computation in datacenters. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (2010), OSDI'10, pp. 75–88.

[40] HERODOTOU, H., DONG, F., AND BABU, S. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing* (2011), SOCC '11, pp. 18:1–18:14.

[41] HORVATH, T., AND SKADRON, K. Multi-mode energy management for multi-tier server clusters. In *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques* (2008), PACT '08, pp. 270–279.

[42] HUANG, C., SIMITCI, H., XU, Y., OGUS, A., CALDER, B., GOPALAN, P., LI, J., AND YEKHANIN, S. Erasure coding in windows azure storage. In *Proc. of the USENIX Conference on Annual Technical Conference* (2012), ATC'12, pp. 2–2.

[43] INTEL CORP. Dataplane Performance Development Kit. `https://dpdk.org`, 2018.

[44] INTEL CORP. Storage Performance Development Kit. `https://01.org/spdk`, 2018.

[45] JONAS, E., PU, Q., VENKATARAMAN, S., STOICA, I., AND RECHT, B. Occupy the cloud: distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017), SOCC'17, pp. 445–451.

[46] KHAN, O., BURNS, R., PLANK, J., PIERCE, W., AND HUANG, C. Rethinking erasure codes for cloud file systems: Minimizing i/o for recovery and degraded reads. In *Proc. of the 10th USENIX Conference on File and Storage Technologies* (2012), FAST'12, pp. 20–20.

[47] KLIMOVIC, A., LITZ, H., AND KOZYRAKIS, C. Reflex: Remote flash == local flash. In *Proc. of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems* (2017), ASPLOS '17, pp. 345–359.

[48] KLIMOVIC, A., LITZ, H., AND KOZYRAKIS, C. Selecta: Heterogeneous cloud storage configuration for data analytics. In *Proc. of the USENIX Annual Technical Conference (ATC'18)* (2018), pp. 759–773.

[49] KLIMOVIC, A., WANG, Y., KOZYRAKIS, C., STUEDI, P., PFEFFERLE, J., AND TRIVEDI, A. Understanding ephemeral storage for serverless analytics. In *Proc. of the USENIX Annual Technical Conference (ATC'18)* (2018), pp. 789–794.

[50] KRIOUKOV, A., MOHAN, P., ALSPAUGH, S., KEYS, L., CULLER, D., AND KATZ, R. H. Napsac: Design and implementation of a power-proportional web cluster. In *Proc. of the First ACM SIGCOMM Workshop on Green Networking* (2010), Green Networking '10, pp. 15–22.

[51] LABS, R. Redis. `https://redis.io`, 2018.

[52] LEE, E. K., AND THEKKATH, C. A. Petal: Distributed virtual disks. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems* (1996), ASPLOS VII, pp. 84–92.

[53] LI, H., GHODSI, A., ZAHARIA, M., SHENKER, S., AND STOICA, I. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proc. of the ACM Symposium on Cloud Computing* (2014), SOCC '14, pp. 6:1–6:15.

[54] LIM, H. C., BABU, S., AND CHASE, J. S. Automated control for elastic storage. In *Proceedings of the 7th International Conference on Autonomic Computing* (2010), ICAC '10, pp. 1–10.

[55] MASHAYEKHI, O., QU, H., SHAH, C., AND LEVIS, P. Execution templates: Caching control plane decisions for strong scaling of data analytics. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference* (Berkeley, CA, USA, 2017), USENIX ATC '17, USENIX Association, pp. 513–526.

[56] MICROSOFT. Azure functions. `https://azure.microsoft.com/en-us/services/functions`, 2018.

[57] MICROSOFT AZURE. Azure redis cache. `https://azure.microsoft.com/en-us/services/cache`, 2018.

[58] MICROSOFT AZURE. SQL data warehouse. `https://azure.microsoft.com/en-us/services/sql-data-warehouse`, 2018.

[59] NARAYANAN, D., THERESKA, E., DONNELLY, A., ELNIKETY, S., AND ROWSTRON, A. Migrating server storage to ssds: Analysis of tradeoffs. In *Proc. of the 4th ACM European Conference on Computer Systems* (2009), EuroSys '09, pp. 145–158.

[60] NETFLIX. Scryer: Netflixs predictive auto scaling engine. `https://medium.com/netflix-techblog/scryer-netflixs-predictive-auto-scaling-engine-a3f8fc922270`, 2013.

[61] NETTO, M. A. S., CARDONHA, C., CUNHA, R. L. F., AND ASSUNCAO, M. D. Evaluating auto-scaling strategies for cloud computing environments. In *Proceedings of the 2014 IEEE 22Nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems* (2014), MASCOTS '14, pp. 187–196.

[62] NGUYEN, H., SHEN, Z., GU, X., SUBBIAH, S., AND WILKES, J. AGILE: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proc. of the 10th International Conference on Autonomic Computing* (2013), ICAC'13, pp. 69–82.

[63] QIAN LI, JAMES HONG, D. D. Thousand island scanner (THIS): Scaling video analysis on AWS lambda. `https://github.com/qianl15/this`, 2018.

[64] QU, C., CALHEIROS, R. N., AND BUYYA, R. Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Computing Surveys 51*, 4 (July 2018), 73:1–73:33.

[65] ROY, N., DUBEY, A., AND GOKHALE, A. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Proc. of the 2011 IEEE 4th International Conference on Cloud Computing* (2011), CLOUD '11, pp. 500–507.

[66] SATHIAMOORTHY, M., ASTERIS, M., PAPAILIOPOULOS, D., DIMAKIS, A. G., VADALI, R., CHEN, S., AND BORTHAKUR, D. Xoring elephants: novel erasure codes for big data. In *Proc. of the 39th international conference on Very Large Data Bases* (2013), PVLDB'13, pp. 325–336.

[67] SEIDEN, S. S. On the online bin packing problem. *J. ACM 49*, 5 (Sept. 2002), 640–671.

[68] SHEN, Z., SUBBIAH, S., GU, X., AND WILKES, J. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing* (2011), SOCC '11, pp. 5:1–5:14.

[69] SRINIVASAN, V., BULKOWSKI, B., CHU, W.-L., SAYYAPARAJU, S., GOODING, A., IYER, R., SHINDE, A., AND LOPATIC, T. Aerospike: Architecture of a real-time operational DBMS. *Proc. VLDB Endow. 9*, 13 (Sept. 2016), 1389–1400.

[70] STRUNK, J. D., THERESKA, E., FALOUTSOS, C., AND GANGER, G. R. Using utility to provision storage systems. In *6th USENIX Conference on File and Storage Technologies, FAST 2008, February 26-29, 2008, San Jose, CA, USA* (2008), pp. 313–328.

[71] STUEDI, P., TRIVEDI, A., PFEFFERLE, J., STOICA, R., METZLER, B., IOANNOU, N., AND KOLTSIDAS, I. Crail: A high-performance i/o architecture for distributed data processing. *IEEE Data Engineering Bulletin 40*, 1 (2017), 38–49.

[72] URGAONKAR, B., PACIFICI, G., SHENOY, P., SPREITZER, M., AND TANTAWI, A. An analytical model for multi-tier internet services and its applications. In *Proc. of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (2005), SIGMETRICS '05, pp. 291–302.

[73] VENKATARAMAN, S., YANG, Z., FRANKLIN, M., RECHT, B., AND STOICA, I. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, 2016), pp. 363–378.

[74] VERMA, A., PEDROSA, L., KORUPOLU, M. R., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-scale cluster management at Google with Borg. In *Proc. of the European Conference on Computer Systems* (Bordeaux, France, 2015), EuroSys'15.

[75] WAJAHAT, M., GANDHI, A., KARVE, A., AND KOCHUT, A. Using machine learning for black-box autoscaling. In *2016 Seventh International Green and Sustainable Computing Conference (IGSC)* (Nov 2016), pp. 1–8.

[76] WIRES, J., AND WARFIELD, A. Mirador: An active control plane for datacenter storage. In *Proc. of the 15th USENIX Conference on File and Storage Technologies* (2017), FAST'17, pp. 213–228.

[77] YADWADKAR, N. J., HARIHARAN, B., GONZALEZ, J. E., SMITH, B., AND KATZ, R. H. Selecting the *best* VM across multiple public clouds: a data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017), SOCC'17, pp. 452–465.

[78] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation* (2012), NSDI'12, pp. 15–28.

[79] ZHANG, Y., PREKAS, G., FUMAROLA, G. M., FONTOURA, M., GOIRI, I., AND BIANCHINI, R. History-based harvesting of spare cycles and storage in large-scale datacenters. In *Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation* (2016), OSDI'16, pp. 755–770.