LambdaObjects: Re-Aggregating Storage and Execution for Cloud Computing

Kai Mast
University of Wisconsin-Madison

Andrea C. Arpaci-Dusseau
University of Wisconsin-Madison

Remzi H. Arpaci-Dusseau
University of Wisconsin-Madison

ABSTRACT

Existing cloud computing (or serverless) architectures provide convenient abstractions for application developers, but exhibit high latencies and do not support strong consistency guarantees. These limitations stem not only from the overheads due to virtualization, but also because storage and compute layers are disaggregated.

We introduce LambdaObjects; a new abstraction for serverless systems where data and compute are co-located. Data is encapsulated into objects, each associated with a set of methods that allow accessing and modifying the data. Functions then execute directly at the nodes the data is stored at. Our early results demonstrate that this architecture provides lower latencies, while sacrificing some elasticity offered by conventional serverless systems. In addition, LambdaObjects provide strong consistency and enable efficient caching mechanisms and fault-tolerance with low overhead.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**.

KEYWORDS

cloud computing, serverless, storage systems

1 INTRODUCTION

Recently, separating storage and compute layers, or even fully disaggregating physical resources within a cluster [9, 35, 42, 51], has become a popular approach for building distributed systems and their underlying infrastructure. Forgoing the requirement to co-locate the various components of a system allows for easier scaling of resources and ensures that failures of one component do not affect other parts of the system.

A common application of disaggregation is serverless environments for which cloud providers, such as Amazon Web Services [48] and Microsoft Azure [39], provide distinct compute and storage layers. Serverless services allow uploading functions, which can then be invoked without explicitly provisioning servers for them to execute on. These functions maintain state by interacting with a storage layer [6, 22, 38]. Each layer abstracts away the underlying physical or virtual machines, ensures sufficient resources are allocated, and ensures failures of individual machines do not disrupt execution or cause data loss.

Even though building applications in a serverless environment is fairly straightforward and allows application developers to sidestep many challenges associated with building distributed systems, the technology has not received as much adoption as one would expect. In fact, most large-scale services still rely on self-hosted storage systems, such as CockroachDB [32] or MongoDB [40], and compute layers that are built from scratch and deployed as conventional macro- or microservices [2, 11].

The low adoption rate of serverless systems has two reasons. First, serverless systems have high start-up latencies due to the use of containers or virtual machines [43, 52]. These isolation mechanisms are necessary to ensure failures, benign or Byzantine [34], of one application do not affect other applications in the same serverless environment. Second, separating storage and compute layers, each with their own scaling and placement mechanisms, makes it hard to provide strong consistency guarantees without a significant impact on performance. While recent work has made progress in reducing startup latencies of serverless environments [3, 5, 13, 29, 43], overcoming the limitations in storage consistency without a significant redesign of how these systems work is much harder to achieve.

Kai Mast, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau

We propose LambdaObjects: an abstraction for building serverless applications where storage and compute are co-located. Similar to object-oriented programming, this abstraction encapsulates data as objects and associates a set of functions with that object. An object's functions can only modify data associated with the object itself, but can invoke functions of other objects. Functions execute where the associated object's data is located, which provides strong consistency with low overhead. Application logic is then implemented as a graph of function calls as in other serverless systems. This object-centered design enables modular modular applications that avoid expensive data transfers in many cases. We also demonstrate how this model enables efficient sharding, fault-tolerance, consistent caching, and linearizability.

Applications written against this abstraction are as easy to develop and deploy as other serverless applications while providing performance close to that of custom microservices and strong consistency. From our early results, we believe this architecture fits many applications that do not require real-time performance (latencies below 1*ms*), but still need to respond to user input quickly, such as most web applications. Strong consistency is a requirement for many projects (e.g., financial applications) and generally eases application development as it is easier to reason about [1]. This, in turn, significantly increases the scope of what can be built as a serverless application, i.e., without manual deployment.

The remainder of this paper gives an overview of this new abstraction, its trade-offs, and early evaluation results. First, we will discuss the current limitations of serverless system the intuition behind LambdaObjects in detail (§2). Then, we specify this new data and compute model in more depth (§3), detail LambdaStore, a system that supports it (§4), and provide early results on how this implementation performs (§5).

2 BACKGROUND AND MOTIVATION

LambdaObjects are designed with applications in mind that require low latencies, high scalability and elasticity, and strong consistency guarantees. We define low latencies as being below 100ms, which is often considered the threshold at which delays become noticeable enough to affect the user experience [8]. Scalable systems are able to increase their throughput without major changes to their architecture. Elastic systems react to workload *changes* quickly by adding or removing resources.

Strong consistency guarantees are important to a broad range of applications. For example, a user might unfriend (or even block) another user and expect that any post they create after this will not be visible to that party. Without, at least, causal consistency [4], the operation blocking the user might be seen by some nodes in the system after they already processed the post. Similarly, an application processing digital

payments requires strong consistency to ensure a transaction reads an up-to-date account balance and, as a result, does not spend more money than is availabe.

In addition, we require an abstraction that makes it easy to create new applications and does not waste physical resources. Keeping the required developer effort low allows adding new features quickly, reduces the maintaince burden, and lowers the number of potential bugs. Ideally, a system always uses all resources it has allocated and frees any unneeded resources, which lowers cost for both the application developer and the cloud provider.

A common example of applications with such requirements are social network or microblogging services, such as Twitter. Retwis [45], which is commonly used to benchmark storage systems, implements a basic microblogging service where users can follow other users and post status messages that propagate to all their followers' timelines. Recent work has shown that even with efficient caching layers and without wide-area communication, conventional serverless systems still exhibit latencies of up 500ms for Retwis [55], indiciating more work is needed.

2.1 Conventional Serverless

Serverless architectures fulfill our requirements of scalablility, elasticity, and resource utilization as show in Table 1. At a high level, they allow developers to upload self-contained functions and abstract away physical or virtual machines used by the application. The stored functions can be invoked by the application's frontend, in which case the cloud provider automatically allocates the required resources. Functions can invoke other functions and application logic is often implemented as a directed acyclic graph of function calls, usually called a *job*. Such cloud programming systems also provide mechanisms to tolerate failures of functions and replicate data automatically.

Unfortunately, serverless systems currently do not meet the requirements for strong consistency and low latencies for two reasons. First, they employ containers or other virtualization technologies to be able to co-locate multiple applications within the same system. This can create a significant delay depending on the underlying technologies. Second, as outlined before, serverless systems are fully disaggregated which adds additional delays due to network communications and makes it hard to achieve strong consistency.

Recent Improvements. Previous work has proposed ways to mitigate the high startup latencies of serverless environments. First, caching or snapshotting virtual machines [3, 13, 58], containers [5, 43], or the language runtime within a container [29], each avoiding significant overheads compared to cold stars. Second, Nightcore [29] converts serverless functions into long-running microservices, which provides better scalability, but slightly weaker isolation guarantees as multiple function calls execute within the same container. Third, recent work

	LambdaObjects	Custom (Micro-)services	Conventional Serverless
Latency	Low (1-10ms)	Very Low (<1ms)	High (>100ms)
Scalability	High	Implementation-Specific	High
Elasticity	Medium	Low	High
Consistency	Strong	Implementation-Specific	Weak
Developer Effort	Low	High	Low
Resource Utilization	High	Low	High

Table 1: Simplified comparison between LambdaObject's aggregated serverless architecture, custom-built (micro-)services, and conventional serverless architectures. Note, that these metrics are also influenced by the type of application and the specific workload.

built serverless systems with lightweight isolation mechanisms such as WebAssembly [53] to reduce startup latencies.

Recent work also showed that caches at the compute layer can alleviate some storage latency and consistency limitations [20, 36, 41, 54, 55]. In addition, Jia et al. [28] demonstrated that consistent reads can be guaranteed using a shared log. However, such mechanisms typically do not provide strong consistency, i.e., linearizability [26]. Disaggregation also makes it hard to provide data locality, which results in frequent cache misses for such approaches. While weaker consistency models or conflict-free data types are convenient and useful for certain applications, they can introduce undesired or even safety-critical inconsistencies and create an additional burden for application developers who have to navigate these more complicated semantics. It is possible to add transactional semantics to conventional serverless environments [14, 60], but due to the limitations mentioned above, such approaches incur high latencies and, for applications with lock contention, low throughput.

2.2 Custom Micro- and Macroservices

Custom implementations are able to provide low latencies and, potentially, strong consistency, but are expensive to build and maintain. Instead of using an abstraction designed by a cloud provider to fit many applications, they are built from the ground up to support a specific application and the expected workload. Additionally, developers can use a storage system in their application that support the desired consistency guarantees. However, designing and building the entire application stack and not just the serverless functions is more time-intensive. Recently such custom systems are more commonly built as microservices, which make each component of the system a separate service that can be developed, tested, and deployed individually [15].

Custom services execute on dedicated resources, which reduces startup delays but harms elasticity and can potentially be costly. For example, a service might consist of a storage node and one node for each of its microservices. This means there is never a cold start as the microservices are constantly running and can respond quickly when contacted.

However, picking such an allocation in advance requires some knowledge about the expected workload and usually results in over-provisioning of resources to be able to tolerate potential high demand. Aside from potentially high cost due to over-provisioning, manual deployment of services is costly. Orchestration tools like Kubernetes [19] can ease deployment, but it is still move involved than a fully automated serverless solution.

2.3 A Re-design of Cloud Programming

The recent improvements to cloud programming are significant but not sufficient to reach our previously outlined goals of latencies below 100ms and strong consistency. Instead of incremental changes, we propose to rethink how serverless applications are written and to redesign the structure of the underlying system. A key observation that guides this redesign is that significant work, e.g., ordering and replication of requests, is replicated within both layers. Reducing this duplication may not only improve performance, but also simplify the design of a cloud computing infrastructure.

Application logic executing close to, or as part of, the storage system are able to overcome limitations in latency and consistency. Here, data can be retrieved quickly as the latency between the compute environment and the storage backend is minimal. Similarly, one can ensure strong consistency as the likelihood of stale data being retrieved is significantly lower. A new data and execution model eases the development for serverless applications for such a co-located architecture.

LambdaObjects aim to provide lower latencies and stronger consistency than conventional serverless and to provide similar ease of development and deployment for application developers and administrators. However, co-locating data and compute harms elasticity as data needs to be migrated when adapting to workload changes, but a sharded datastore is still inherently scalable. Additionally, a new data model makes it harder for existing codebases to be ported to this abstraction, especially those requiring a POSIX-style API. Finally, the LambdaObjects model must still allow sharing resources among applications to fully utilize the available resources.

To summarize, Table 1 outlines how the LambdaObjects abstraction aims to be a compromise between serverless and

custom implementations suitable for most applications. Conventional serverless environments will most likely still be more efficient for workloads with high variability or with infrequent data-accesses, for example continuous integration tasks or machine learning, and custom implementations are still necessary for applications that require very low latencies. In any case, which approach is the best depends highly on the type of application and workload.

3 LAMBDAOBJECTS

LAMBDAOBJECTS are intended to implement a small piece of functionality, e.g., a user authentication mechanism, that is part of a larger application, e.g., an online store. Each of these components can then be built and tested individually like conventional microservices. Objects can then invoke methods of other objects to compose more complex application logic. However, unlike microservices, object method invocations are short-lived and isolated from other invocations of the same method.

Objects have access to their associated storage through a key-value API and some utility functions in addition to executing and invoking functions. This minimal API ensures a small attack surface and supports different application types and data models. For example, some applications may want a storage system that provides access to raw, unstructured data and others might require an abstraction more akin to SQL. In the latter case packing and unpacking data can be implemented within the runtime.

To ease application development, we introduce the notion of *object types*. Each object type holds a set of functions in a format specific to the implementation, e.g., as ELF binaries. Object types also contain a set of *fields*, which are either a single opaque piece of data or collection of data entries indexed by a key. Objects can then be instantiated from these types.

3.1 Consistency Model

LAMBDAOBJECTS support *invocation linearizability*, which we informally define as follows. First, data accesses and modifications within a single function invocation are atomic; either all succeed or none. Second, function invocations are isolated; partial writes of one invocation are not visible to other function invocations. Third, function invocations provide "real-time" guarantees; if a functions call is successful, it is guaranteed that all following function calls will see its modifications.

These guarantees do not span across function calls to avoid cyclic dependencies and aborts. In our current implementation, this means that invoking one function from another will commit all changes of the first call before performing the second. Thus, if a function f invokes another function f' in the middle of its execution, the parts of f before and after invoking f' will be treated as two separate function invocations

```
1  type User:
2    field name: String
3    field followers: List<ObjectId>
4    field timeline: List<(String, Time, String)>
5
6    pub func create_post(self, msg):
7      let time = get_time()
8      self.store_post(self.name, time, msg)
9
10     for oid in self.followers:
11       get_object(oid)
12         .store_post(self.name, time, msg)
13
14   pub func get_timeline(self, limit):
15     let result = []
16     let iter = self.timeline.iterate()
17     for result.len() < limit and iter.has_more():
18       result.push(iter.next())
19     return result
20
21   func store_post(self, src, time, msg):
22     self.timeline.push((src, time, msg))
```

Listing 1: Implementation of ReTwis in pseudocode. Low-level API calls are hidden behind a higher-level abstraction Functionality that creates accounts or adds followers is not shown.

in the context of consistency. While similar to snapshot isolation [10] at a first glance, this is a slightly weaker guarantee as writes commit at the end of each function invocation not at the end of the job.

We envision that future versions of the LAMBDAOBJECTS model will support serializable transactions [44] spanning multiple function calls, but this is out of the scope of this workshop paper. Conveniently, embedding execution into the database itself allows using proven transaction processing protocols from existing database management systems instead of having to develop an entirely new mechanism.

3.2 Application Example

We can implement the microblogging application from Section 2 using LAMBDAOBJECTS in a straightforward way. Each User object holds a list of their posts, a set of all their followers, as well as a timeline containing posts of all user's they follow. Listing 1 outlines a potential implementation of a User object and its create_post method. The code omits blocking/following users and other functionality due to limited space.

create_post first generates a new post and the stores it in the user's and all its followers' timelines. Updating many follower timelines at once is done quickly by running the store_post calls in parallel (not shown in the pseudocode). Invocation linearizability prevents aborts due to concurrency, so that requests will be processed quickly. This consistency

property also ensures that causality is respected so that, for example, blocked users will be removed from the follower list before the new posts can be generated.

4 PROTOTYPE IMPLEMENTATION

4.1 Conventional Serverless Architectures

Before we discuss the design of a system that supports the LambdaObjects model, we outline how regular serverless architecuters, such as OpenWhisk [17], work. Here, clients (or client-facing frontends) interact with the compute layer through a load balancer that distributes computation. This load balancer must also log client requests in a durable way to ensure that, in case of compute node failures, there will always be a response generated [50]. For example, in the case of OpenWhisk this replication mechanism is implemented using Apache Kafka [18].

Compute nodes then interact with the storage backend over the network. Thus, unless there is caching or a transactional layer put in place, each storage access requires a network round-trip. The storage backend itself is replicated to ensure durability of data. If a lambda function invokes other lambda functions during their execution, they will contact the load-balancer again, introducing another round of indirection. This avoids re-executing the entire workload if a single function fails at the cost of increased latency.

4.2 LambdaStore Architecture

We implemented LambdaStore: a storage system that supports execution of methods compiled to WebAssembly [24]. We chose WebAssembly because, as previous work showed [53, 61], it has low overhead and allows embedding untrusted code directly into a process without sacrificing security. However, a similar design could be achieved by placing containers or virtual machines executing conventional binaries on the same node as the relevant storage process.

WebAssembly provides software-based isolation and metering. Runtimes compile untrusted code into trusted machine code by adding safeguards to every memory access. Additional checks can be added to limit the amount of computation a function invocation is allowed to perform. The resulting machine code can then be executed at almost native speed.

Storage nodes leverage the properties of LambdaObjects in two distinct ways. First, they represent the lowest form of concurrency. Because functions only directly access data within the same object, nodes can avoid write conflicts by not scheduling two functions modifying data of the same object at the same time. The abstraction, thus, allows the application developer to determine the granularity of locks. LambdaStore then combines function scheduling and concurrency control.

Second, objects are microshards [7]. Because their content is self-contained, they can be migrated by themselves without causing disruption to computation involving other objects. Microsharding, unlike, e.g., hash-based sharding [47, 56], allows maintaining data locality. The LambdaObjects abstraction enables application developers to define what data "belongs together" in a straightforward way.

4.2.1 Replicating LambdaObjects. Replication is added to this design in a straightforward way. We chose primary-backup [12] as it provides low-latencies compared to, e.g., chain replication [57]. Functions that modify data are executed at the primary and the results of the computation are replicated to the backup nodes. Read-only functions can execute at any replica to increase throughput.

Fault-tolerance is ensured through a cluster-wide coordination service. This service is replicated using Paxos [33] to ensure availability at all times. If a node fails, the coordinator will reconfigure the affected shards and notify all participants. Clients, or nodes, waiting for a response from that shard will reissue their request if needed. Thus, a function invocation results in at most one network round-trip within the responsible replica set. This design also obviates the need for an additional logging service and scales well as the coordinator is only involved during reconfigurations.

4.2.2 Consistent Caching. Even an efficient implementation of a function execution environment can cause significant overhead when invoked frequently, which can be avoided using caching. Caching computation results in an environment where data and computation are not co-located can easily result in inconsistencies when caches are not up-to-date. However, in a co-located setting, storage nodes always have access to the most recent state of the data.

For deterministic read-only functions, the storage system allows caching results consistently. Here, storage nodes merely record the output of a function, a hash of its input, and its read set in the forms keys and value hashes. Nodes then only re-execute such functions if the input or reads have changed. This is similar to how MySQL handles deterministic stored procedures [27].

5 PRELIMINARY EVALUATION

We compare our aggregated architecture against a disaggregated design with separate compute and storage layers. For both designs, we use WebAssembly as our isolation mechanism to make the comparision fair. The disaggregated variant is implemented as a standalone process executing WebAssembly binaries. In addition, the baseline uses our prototype as its storage layer to ensure that implementation details of the database itself does not skew the results. In both cases LambdaStore uses LevelDB [21] to persist data.

We allocate one machine for compute and three machines for storage. The storage machines form a replica set and do not
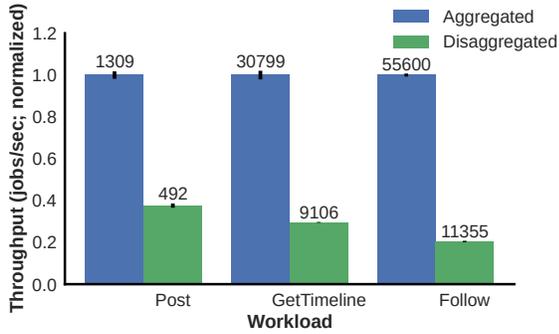
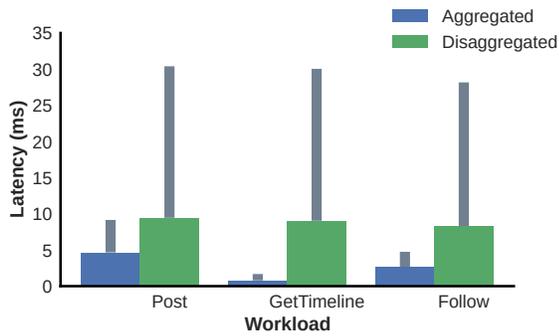**Figure 1: Normalized throughput of the ReTwis benchmark.**

Figure 2: Latencies of the ReTwis benchmark. Big bars show median values and small bars the 99th percentile.

perform sharding. In the disaggregated variant, functions execute on a dedicated compute node separate from the storage nodes. For the aggregated variant, functions directly execute at the primary storage node. In either case, clients directly contact the executing node and there is no load balancer or frontend. The experiment is run on Cloudlab [16], where each machine is equipped with two Intel Xeon® Silver 4114 CPUs totalling 20 physical cores and 188 GiB of memory. All machines are in the same physical location, and we do not simulate a wide-area network.

We evaluated three different kinds of workloads for the application described in Section 3: *Post* creates a new post and updates all timelines of the account's followers, *GetTimeline* is a read-only task that returns the timeline for a specific user, and *Follow* adds a new follower to an account. For the aggregated variant, we enforce invocation linearizablity as outlined before, while the disaggregated variant provides no consistency guarantees. We set up 10,000 accounts and run up to 100 concurrent client requests for all workloads, which we found to yield the maximum throughput.

Figure 1 plots the total throughput of these workloads and Figure 2 the latencies. The aggregated variant shows an increase of at least 160% for throughput and a decrease of at least 50% for median latency. Because all machines are located within the same cluster, and because we do not add artificial

network delays, latencies are generally low. We also observe a higher variance in latencies for the disaggregated baseline even when lowering the number of concurrent requests. A single job in the *Post* workload requires multiple function calls, the initial function call and one for each follower, which results in lower throughput compared to the other workloads.

6 RELATED WORK

Previous work proposed embedding computation in the storage layer. Many database management systems provide mechanisms for stored procedures [23, 27], which usually only support functions written in SQL not arbitrary binaries. Active storage [46] and Willow [49] allow executing application code within the storage device itself, e.g., on an SSD's microprocessor, but do not provide means to replicate or shard such data. Biscuit [25] and YourSQL [30] allow executing parts of a single application directly at the storage nodes. Zhang et al. [61] first proposed embedding WebAssembly into the storage layer, but their work did not address replication or sharding of data. Finally, blockchains embed untrusted binaries (usually compiled to custom bytecode or WebAssembly) into their storage processes to support "smart contracts" [59].

Recent work also improved how data is passed between serverless function invocations of the same job. Pocket [31] provides an ephemeral storage layer to pass data between functions efficiently. SONIC [37] decides between data passing strategies depending on the location of the other function(s) and the type of fanout, e.g., single, scatter, or broadcast.

7 CONCLUSION AND OPEN PROBLEMS

We introduced LAMBDAOBJECTS: a new abstraction to build efficient and low-latency serverless applications. This paper demonstrated how to build applications using this abstraction and gave insights on how the underlying system architecture improves upon conventional serverless architectures. Our early results show promising performance improvements over the disaggregated design. Future work has to investigate how to efficiently shard and scale systems that support LAMBDAOBJECTS so that they provide similar elasticity guarantees as other serverless systems. Additionally, one will need to add consistency guarantees for transactions spanning multiple function calls.

REFERENCES

[1] Mike Curtiss (Google). Why you should pick strong consistency, whenever possible. https://cloud.google.com/blog/products/databases/why-you-should-pick-strong-consistency-whenever-possible (Last Accessed May 2022).

[2] Mazdak Hashemi (Twitter). The Infrastructure Behind Twitter: Scale. https://blog.twitter.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale (Last Accessed March 2022).

[3] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. *Symposium on Networked System Design and Implementation,* pages 419-434, Santa Clara, California, February 2020.

[4] Mustaque Ahamad, Gil Neiger, Jame E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing,* 9(1):37–49, 1995.

[5] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-Performance Serverless Computing. *USENIX Annual Technical Conference,* pages 923-935, Boston, Massachusetts, July 2018.

[6] Amazon. S3 Cloud Object Storage. https://aws.amazon.com/s3/ (Last Accessed March 2022).

[7] Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, David Nagle, and Michael Stumm. Sharding the Shards: Managing Datastore Locality at Scale with Akkio. *Symposium on Operating System Design and Implementation,* pages 445-460, Carlsbad, California, October 2018.

[8] Christian Attig, Nadine Rauh, Thomas Frank, and Josef F. Krems. System latency guidelines then and now–is zero latency really considered necessary? *International Conference on Engineering Psychology and Cognitive Ergonomics,* pages 3–14, 2017.

[9] Frank Bell, Raj Chirumamilla, Bhaskar B. Joshi, Bjorn Lindstrom, Ruchi Soni, and Sameer Videkar. How Snowflake Compute Works. *Snowflake Essentials,* pages 223–237, 2022.

[10] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A Critique of ANSI SQL Isolation Levels. *SIGMOD International Conference on Management of Data,* pages 1-10, San Jose, California, May 1995.

[11] Netflix Technology Blog. Netflix Platform Engineering — we're just getting started. http://netflixtechblog.com/neflix-platform-engineering-were-just-getting-started-267f65c4d1a7 (Last Accessed March 2022).

[12] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The primary-backup approach. *Distributed systems,* 2:199–216, 1993.

[13] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. SEUSS: skip redundant paths to make serverless fast. *European Conference on Computer Systems,* pages 32-1, Heraklion, Greece, April 2020.

[14] Martijn de Heus, Kyriakos Psarakis, Marios Fragkoulis, and Asterios Katsifodimos. Distributed transactions on serverless stateful functions. *DEBS '21: The 15th ACM International Conference on Distributed and Event-based Systems, Virtual Event, Italy, June 28 - July 2, 2021,* pages 31–42, 2021.

[15] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering,* pages 195–216, 2017.

[16] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuang-Ching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet,

Snigdhaswin Kar, and Prabodh Mishra. The Design and Operation of CloudLab. *USENIX Annual Technical Conference,* pages 1-14, Renton, Washington, July 2019.

[17] Apache Software Foundation. OpenWhisk. https://openwhisk.apache.org/ (Last Accessed March 2022).

[18] Apache Software Foundation. OpenWhisk Architecture. https://cwiki.apache.org/confluence/display/OPENWHISK/System+Architecture (Last Accessed March 2022).

[19] Linux Foundation. Kubernetes. https://kubernetes.io/ (Last Accessed May 2022).

[20] Alexander Fuerst and Prateek Sharma. FaasCache: keeping serverless computing alive with greedy-dual caching. *International Conference on Architectural Support for Programming Languages and Operating Systems,* pages 386-400, Virtual, Anywhere, April 2021.

[21] Sanjay Ghemawat and Jeff Dean. LevelDB. https://github.com/google/leveldb (Last Accessed May 2022).

[22] Google. Google Cloud Storage. https://cloud.google.com/storage/ (Last Accessed March 2022).

[23] The PostgresSQL Global Development Group. PostgresSQL Stored Procedures. https://www.postgresql.org/docs/current/sql-createprocedure.html (Last Accessed May 2022).

[24] W3 WebAssembly Working Group. WebAssembly Specification. https://webassembly.org/specs/ (Last Accessed March 2022).

[25] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. Biscuit: A Framework for Near-Data Processing of Big Data Workloads. *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016,* pages 153–165, 2016.

[26] Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems,* 12(3):463-492, 1990.

[27] Oracle Inc. MySQL Stored Procedures. https://dev.mysql.com/doc/refman/8.0/en/create-procedure.html (Last Accessed May 2022).

[28] Zhipeng Jia and Emmett Witchel. Boki: Stateful Serverless Computing with Shared Logs. *Symposium on Operating Systems Principles,* pages 691-707, Koblenz, Germany, November 2021.

[29] Zhipeng Jia and Emmett Witchel. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. *International Conference on Architectural Support for Programming Languages and Operating Systems,* pages 152-166, Virtual, Anywhere, April 2021.

[30] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. YourSQL: A High-Performance Database System Leveraging In-Storage Computing. *Proceedings of the VLDB Endowment,* 9(12):924-935, 2016.

[31] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic Ephemeral Storage for Serverless Analytics. *Symposium on Operating System Design and Implementation,* pages 427-444, Carlsbad, California, October 2018.

[32] Cockroach Labs. CockroachDB Documentation. https://www.cockroachlabs.com/docs/stable/ (Last Accessed March 2022).

[33] Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems,* 16(2):133-169, 1998.

[34] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems,* 4(3):382-401, 1982.

[35] Sergey Legtchenko, Hugh Williams, Kaveh Razavi, Austin Donnelly, Richard Black, Andrew Douglas, Nathanael Cheriere, Daniel Fryer, Kai Mast, Angela Demke Brown, Ana Klimovic, Andy Slowey, and Antony I. T. Rowstron. Understanding Rack-Scale Disaggregated Storage. *Workshop on Hot Topics in Storage and File Systems,* Santa

Clara, California, July 2017.

[36] Taras Lykhenko, Rafael Soares, and Luis Rodrigues. FaaSTCC: efficient transactional causal consistency for serverless computing. *ACM/IFIP Middleware Conference,* pages 159-171, Quebec City, Canada, November 2021.

[37] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. SONIC: Application-aware Data Passing for Chained Serverless Applications. *USENIX Annual Technical Conference,* pages 285-301, Virtual, Anywhere, July 2021.

[38] Microsoft. Azure Blob Storage. https://azure.microsoft.com/en-us/services/storage/blobs/ (Last Accessed March 2022).

[39] Microsoft. Azure Functions. https://azure.microsoft.com/en-us/services/functions/ (Last Accessed March 2022).

[40] MongoDB. MongoDB Documentation. https://www.mongodb.com/docs/ (Last Accessed March 2022).

[41] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. OFC: an opportunistic caching system for FaaS platforms. *European Conference on Computer Systems,* pages 228-244, Edinburgh, Scotland, April 2021.

[42] Mihir Nanavati, Jake Wires, and Andrew Warfield. Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage. *Symposium on Networked System Design and Implementation,* pages 17-33, Boston, Massachusetts, March 2017.

[43] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. *USENIX Annual Technical Conference,* pages 57-70, Boston, Massachusetts, July 2018.

[44] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM,* 26(4):631-653, 1979.

[45] Redis. ReTwis Documentation. https://redis.io/docs/reference/patterns/twitter-clone/ (Last Accessed March 2022).

[46] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. Active Storage for Large-Scale Data Mining and Multimedia. *International Conference on Very Large Data Bases,* pages 62-73, New York, New York, August 1998.

[47] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. *ACM/IFIP Middleware Conference,* pages 329-350, Heidelberg, Germany, November 2001.

[48] Amazon Web Services. AWS Lambda. https://aws.amazon.com/lambda/ (Last Accessed March 2022).

[49] Sudharsan Seshadri, Mark Gahagan, Meenakshi Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A User-Programmable SSD. *Symposium on Operating System Design and Implementation,* pages 67-80, Broomfield, Colorado, October 2014.

[50] Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. *USENIX Annual Technical Conference,* pages 205-218, Virtual, Anywhere, July 2020.

[51] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. *USENIX Annual Technical Conference,* Renton, Washington, July 2019.

[52] Mikhail Shilkov. Comparison of Cold Starts in Serverless Functions across AWS, Azure, and GCP. https://mikhail.io/serverless/coldstarts/big3/ (Last Accessed May 2022).

[53] Simon Shillaker and Peter R. Pietzuch. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. *USENIX Annual Technical Conference,* pages 419-433, Virtual, Anywhere, July 2020.

[54] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E. Gonzalez, Joseph M. Hellerstein, and Jose M. Faleiro. A fault-tolerance shim for serverless computing. *European Conference on Computer Systems,* pages 15-1, Heraklion, Greece, April 2020.

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

[56] Ion Stoica, Robert Tappan Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Conference,* pages 149-160, San Diego, California, August 2001.

[57] Robbert van Renesse and Fred B. Schneider. Chain Replication for Supporting High Throughput and Availability. *Symposium on Operating System Design and Implementation,* pages 91-104, San Francisco, California, December 2004.

[58] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. FaaSNet: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute. *USENIX Annual Technical Conference,* pages 443-457, Virtual, Anywhere, July 2021.

[59] Gavin Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger. *Ethereum Yellowpaper,* 2014.

[60] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. *Symposium on Operating System Design and Implementation,* pages 1187-1204, Banff, Canada, November 2020.

[61] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the Gap Between Serverless and its State with Storage Functions. *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019,* pages 1–12, 2019.