Rendezvous: Where Serverless Functions Find Consistency

Mafalda Sofia Ferreira
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa

João Ferreira Loff
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa

João Garcia
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa

Abstract

Function-as-a-Service has been notoriously difficult to compose and integrate, leaving developers to resort to third-party datastores and queues as means to share state between functions. Applying this architecture on a geo-replicated setting may produce inconsistent application states due to replication delays and weak consistency guarantees, thereby resulting in inconsistent function outputs and ultimately a bad end-user experience. Rendezvous is a framework that automatically ensures consistency of data in serverless applications using datastores, and guarantees that no stale values are read.

CCS Concepts: • **Software and its engineering** → **Consistency**; • **Computer systems organization** → **Cloud computing**; • **Information systems** → *Cloud based storage*; **Remote replication**.

Keywords: Cloud Computing, Function-as-a-Service, FaaS, Serverless, Consistency, Replication, Cloud Storage Systems

ACM Reference Format:
Mafalda Sofia Ferreira, João Ferreira Loff, and João Garcia. 2023. Rendezvous: Where Serverless Functions Find Consistency. In *4th Workshop on Resource Disaggregation and Serverless (WORDS '23), October 23, 2023, Koblenz, Germany.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3605181.3626290

1 Problem Statement

Function-as-a-Service platforms provide cloud-based applications [4, 13, 21] with an elastic and scalable environment where applications composed of sequences of stateless functions can be dynamically executed in response to application events [18].

In principle, functions execute without state and independently of each other. However, a key open challenge within the serverless community is the direct communication between functions, with multiple approaches to address that issue [11, 17, 27, 30]. Given the limited adoption of these solutions by cloud providers, developers continue to rely on supplementary storage layers such as standard datastores [2, 5], caches [3], or message queues [6, 14] for sharing state between functions.

Developers often use these storage layers in geo-replicated settings with the expectation that written data will be readily available to subsequent functions in different global regions, overlooking the consistency semantics of these datastores' replication algorithms [10, 12]. As developers write into one replica of a service, the replication of that data to other data centers – specially ones in different regions – introduces propagation latency that developers don't always account for [26], and hence may observe stale information. Consequently, the interaction between different functions that access potentially stale data, might result in inconsistencies across their execution.

To gain a better understanding of how these inconsistencies can occur, consider the example application in Fig. 1 that depicts the workflow of a single request in a serverless application. The example explores a Post-Notification application [20] in which users upload new posts, and subscribers receive notifications about new content. The application is composed of two functions: a writer and a reader. The writer executes, e.g. in Europe (EU), and receives as a parameter the post to write and the poster's identifier. Then, it writes the post's data and enqueues the corresponding notification to the followers. The reader is triggered when a notification is delivered to a follower, e.g. in the United States (US). Both functions have access to a replicated storage service containing all posts, and the communication between them is done through a messaging service with queues replicated across both regions. The outline of a typical workflow is as follows:

① A user requests to upload a new post which triggers a new writer function invocation.
② The writer function writes the post's data in $write_P$ to the underlying datastore in EU. The data is then asynchronously propagated to US.
③ The post's identifier p_{id} returned by the write operation is then included in the notification event that is queued by the writer function in $write_N$.

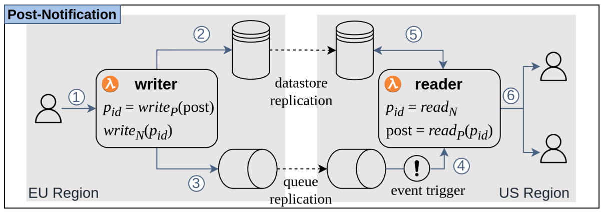

Figure 1. Example of a generic request workflow in the Post-Notification application [20].

④ Upon replication to the US, the reader is triggered by an event from which the post's identifier is extracted in $read_N$.
⑤ The reader fetches the post's content from the datastore in the US region in $read_P(p_{id})$.
⑥ Finally, the notification and the post contents are delivered to the original poster's followers.

The error may be observed in step ⑤: if the notification reaches US prior to the post becoming visible there, the reader function will retrieve the post ahead of time, resulting in absent or outdated content. This reflects in additional incorrect executions when forwarding the notification to subscribers. This example illustrates how the behavior of an application that works at global scale, has to take into account the latency of datastore propagation across regions. These phenomena, described by [20] as *cross-service inconsistencies*, also have an impact at the serverless level when different functions interact across different regions.

In order to assess the impact of these inconsistencies within AWS, we looked at the findings of Antipode [20] for the Post-Notification application, represented in Fig. 1. The experiment employs various combinations of both the post data store and the notification queue. Table 1 details the observed inconsistencies, and how different combinations of datastores result in different percentages of observed inconsistencies. For instance, we can clearly see that using SNS together with almost any post datastore results in a high percentage of cross-service inconsistencies, which attests to how optimized SNS geo-replication is in comparison with post datastores.

Despite this evidence, cloud providers currently do not offer a way for developers to ensure a consistent view within their functions. Consequently, they must face the consequences of these inconsistencies that negatively impact their applications and, consequently, their users.

2 Challenges

There are three significant challenges to preventing inconsistencies in serverless environments.

Geo-replicated Datastores. Serverless applications often rely on geo-replicated datastores [2, 3, 5] to share state between functions [25]. Due to the heavy preference for availability, strong consistency is sacrificed, with developers often settling for eventual consistency semantics [8, 10, 12, 29]. Therefore, data objects are not guaranteed to be immediately replicated, and functions may be unaware of potential inconsistencies.

		post datastore			
		MySQL	DynamoDB	Redis	S3
queue	SNS	95%	95%	88%	100%
	AMQ	8%	7%	13%	100%
	DynamoDB	0%	0%	0%	13%

Table 1. Percentage of observed inconsistencies for several combinations of off-the-shelf AWS services [20].

Lack of Coordination between Functions. Existing alternatives that provide indirect function communication [5, 14] implicitly introduce dependencies between them. The lack of coordination makes it difficult to determine whether data written by preceding and distinct function executions is already visible to subsequent ones.

Restriction of Datastore Modifications. Cloud providers do not offer ways to modify the internal codebase of datastores, which results in two further limitations: (i) they are unable to accommodate the insertion and retrieval of additional metadata, and (ii) they do not offer developers efficient and flexible detection mechanisms to detect changes in datastore objects[1].

3 Rendezvous

We present Rendezvous, a framework targeted at cloud providers that enhances FaaS with a layer that automatically enforces a consistent cross-function view of application data. In short, Rendezvous verifies existing `write` and `read` operations within functions in order to provide a consistent view of all the application's data across multiple datastores while preserving their underlying consistency semantics.

3.1 Overview

Rendezvous is built upon two main concepts: *requests* and *branches*. A request is initiated from an external source (e.g. a user request) which span multiple actions across the application, including function executions and datastore operations. Branches refer to the ramifications arising from a request's actions. Each branch represents an application's execution within a single datastore and across one or more regions. For instance, in Fig. 1, the write post operation is initiated in EU and propagated to US, causing this replication mechanism to open a branch at two regions from a single write operation.

Branches store their identifier, their region(s), the datastore, and their current status (`OPENED` or `CLOSED`) that identify the visibility of data in a datastore region represented by the branch. Branches are opened upon initial creation and

[1]Although, for instance Mongo provides a mechanism that allows developers to hook into replication events [24], these types of features are typically not available in other datastores.

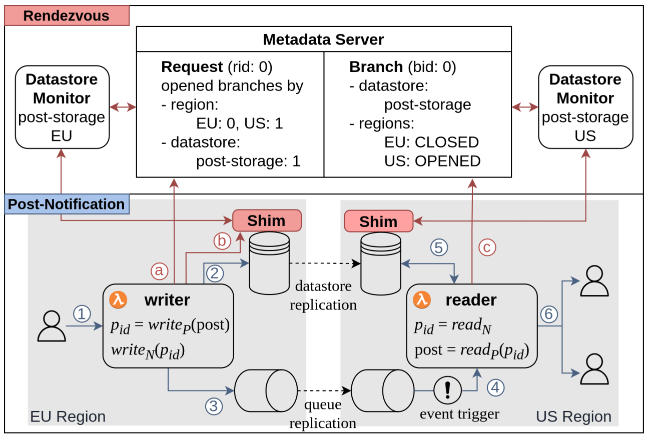

Figure 2. Example of Post-Notification application and respective integration with Rendezvous.

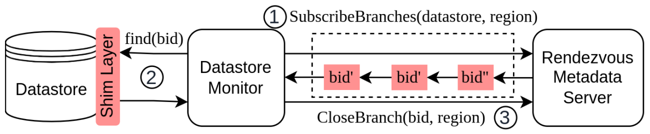
① SubscribeBranches(datastore, region)
find(bid)
②
Datastore
Shim Layer
Datastore Monitor
bid'
bid'
bid''
Rendezvous Metadata Server
CloseBranch(bid, region) ③

Figure 3. Datastore monitor execution and Rendezvous metadata flow during the lifetime of branching subscription stream with a metadata server.

closed when the objects, whose write operation represents these branches, are visible at a specific region.

Requests contains branches whose status (opened or closed) is tracked in relation to four contexts: globally, by datastore, by region, and by both datastore and region. This is exemplified in the data structure in Fig. 2. The request is considered as completed when no branch is left open for a given context. In practice, we delay the functions' read operations until preceding writes from the same request are completed, waiting for (geo-)replication to finish and reestablishing a consistent state (signaled by the closing of its branch at all regions).

For example, in the Post-Notification inconsistency example, Rendezvous corrects the inconsistency by:

ⓐ Registering the branch for the write post operation for EU and US regions.
ⓑ The write operation to the datastore is extended to include additional metadata identifying the corresponding branch.
ⓒ Subsequent reads are blocked until Rendezvous guarantees replication has concluded, and all regions are consistent.

3.2 Architecture

Rendezvous is composed of four core components: (i) a metadata server that monitors the overall progress of requests (ii) a shim layer (tailored to each datastore) that extends client calls with additional Rendezvous metadata, (iii) a Rendezvous API library for applications to call the metadata server and the shim layer, and (iv) a dedicated datastore monitor for each datastore and region, that monitors data replication and updates.

Metadata Server. The branching information is stored in *requests*, *branches* and *subscribers* structures maintained by the metadata server that tallies *opened* and *closed* branches for each request. Meanwhile, the *subscribers* data structures are used to create queues of new branch identifiers to be forwarded to the datastore monitors (see ahead).

Shim Layer. The shim layer replaces and extends original datastore operations through a *bolt-on* approach [9, 20] hence preserving the underlying consistency semantics, while transparently including Rendezvous metadata in each call. The metadata containing branch identifiers is included in write operations and later replicated to any other datastore node in other regions, where its presence ensures that the post is visible to subsequent reads. In the Post-Notification example, when attempting to write the post's content, ② is replaced by a call to the shim layer at ⓑ. The branch identifier is propagated alongside the post to be later detected in US. This mechanism supports the detection of replication of new objects to allow closing branches previously opened by write operations at other regions.

API: Opening Branches. Prior to performing a datastore write, it is necessary to register a new datastore branch at the metadata server, by specifying the targeted datastore. This branch is set as opened for all regions to which the write will be eventually replicated. For instance, in Fig. 2, at ⓐ, a branch for the write post is registered for both regions by invoking the `RegisterBranch(rid=0, datastore=post-storage, regions=[EU,US])`. Afterward, the metadata server creates and opens the branch, sets an `OPENED` status for both EU and US regions, and returns a unique branch identifier, *bid*.

API: Wait for Branch Closing. Synchronization between functions is the key aspect of Rendezvous. Their execution is coordinated through a waiting call that blocks while data from previous write operations is not yet visible in the reader's region. In our system, this is translated into a branch describing the replication of a write operation. This mechanism is complemented by a datastore monitor, which is responsible for automatically closing branches (detailed ahead). Note that, in the event of a wait on multiple writes onto the same datastore, a function must explicitly register the branch that encompasses all the branches for those writes and close it upon completion. This allows metadata servers to identify which branch registrations must be awaited prior to executing the core `wait` logic.

In Fig. 2, the synchronization is achieved by invoking the remote `WaitRequest(rid=0, datastore=post-storage, region=US)` at ⓒ. The call blocks while branches are opened in the *post-storage* node of US, ensuring that the reader will only attempt to read the post when it is guaranteed to be available, maintaining a consistent execution.

Datastore Monitor. Ideally, each datastore should send an event to the metadata server when a replication update for a specific branch identifier arrives at the local region. Although some datastores offer ways to *react-to-replication* [2, 5, 24], this type of mechanism is not commonly available.

Hence, a datastore monitor process was implemented for each local datastore that subscribes to publications provided by the metadata server for every newly added branch (see Fig. 3). When the datastore monitor is notified regarding a new branch identifier (*bid*), it monitors the datastore for its visibility in the region, and, as soon as the identifier is visible, it notifies the metadata server to close the branch with a `CloseBranch(bid, region)` call. We minimize the performance impact of this long-lived connection with the metadata server by using a subscription-based stream, which is only triggered when fresh branching information arrives.

Fig. 2 illustrates how this process is employed. In the example, there are two processes, one for each region (EU and US). In the US, the process is notified by the metadata server whenever a new identifier is created from the new registered branch at ⓐ. Next, it begins to monitor the visibility of the post and its identifier (both written at ⓑ). The process closes the branch for that region by invoking `CloseBranch(bid=0, region=US)`, ensuring that all branches are closed in US. As a result, the blocking `WaitRequest` is released at ⓒ, thereby allowing the reader to safely fetch the post at ⑤ – without risk of observing any inconsistent state.

3.3 Integration with Cloud Platforms

We intend for Rendezvous to be integrated by existing cloud providers as part of their platforms that manage functions, in a way that is completely transparent to the developer. In fact, the only information provided by the developer would be an acknowledgment at configuration time, that Rendezvous would be enabled within a specific function. After this, cloud platforms would take care of wrapping read and write calls to datastores through our shim layer, and would also enhance interactions with Rendezvous-related metadata. More concretely, write operations would be wrapped to register new branches and inject metadata within the write to the datastore. Read operations would automatically perform a `WaitRequest` call which would always ensure a consistent read by the underlying function. Both these calls would transparently contact the metadata server to check on requests and branches status. We also envision that developers could have the flexibility to tune the configuration according to their needs. A key example of this could be seen in `WaitRequest`, where developers could relax the consistency view over the datastores and explicitly bypass the call, effectively trading correctness for performance.

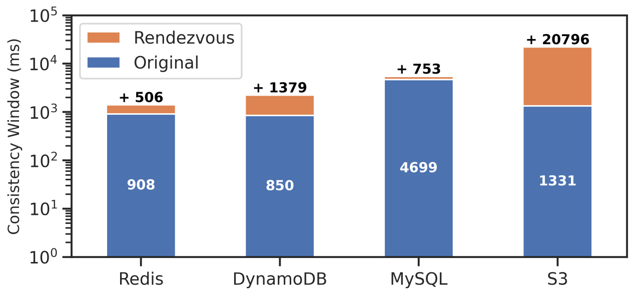

Figure 4. Variation of consistency window for each datastore with and without Rendezvous.

3.4 Optimizations

Rendezvous places one metadata server node in each region where functions are executed. This allows functions to have local access, therefore making Rendezvous-related calls always local. We detail our replication strategy in §3.5.

We also allow developers to bypass the default blocking behavior of `WaitRequest` by either using a timeout (waiving correctness) or by using an asynchronous callback when the data is consistent, into a later point of their functions.

3.5 Replication and Fault Tolerance

The metadata server nodes are asynchronously replicated between regions. This replication model boasts reduced overhead for Rendezvous calls and allows our system to remain operational during network partitions, particularly for branching registrations, albeit with the drawback extending wait requests.

The metadata server takes advantage of a call context to validate its connection with the datastore monitors to detect potential failures. If invalidated by an error or crash on the datastore monitor side, the metadata server promptly closes the connection and retains the remaining identifiers in the queue for an eventual reconnection. We envision this to be further extended and supported by cloud providers, serving as a means to provide real-time alerts and datastore monitor recovery strategies in response to these failures.

4 Evaluation

We evaluated Rendezvous in order to assess the level of performance and effectiveness. Our analysis concerns two key factors: (1) its consistency window, and (2) its scalability.

4.1 Experimental Setup

Similarly to Fig. 1, we deployed the Post-Notification application [20] in AWS using a pipeline of two Lambda functions, placing the writer in EU and the reader in US. For the notification queue, we used AWS SNS with notifications objects flowing from the writer to the reader. We used four different backends as the post-storage for each deployment: Redis, DynamoDB, MySQL, and S3, and all of them were globally replicated across EU and US. For Rendezvous, we deployed

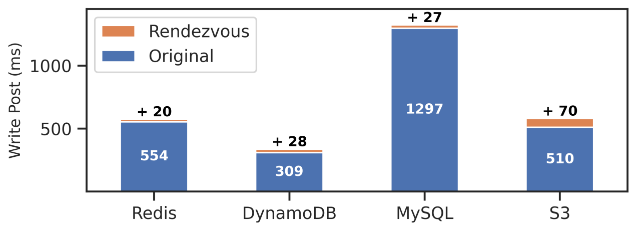

Figure 5. Time spent writing the post between Original vs. Rendezvous, where metadata is included in the write operation.

one server per region, provisioned in AWS EC2 `t2.xlarge` instances with 4 vCPU and 16 GiB RAM. Each server ran the metadata server and the corresponding datastore monitor, with asynchronous replication between regions. For each evaluation run, we configured 1000 different posts, which triggered a corresponding number of writer Lambdas.

Since we used AWS as our cloud provider, we had to emulate Rendezvous's behavior by placing additional calls from Fig. 2 in the corresponding functions. Both ⓐ and ⓑ were placed in the writer, while ⓒ, was placed in the reader immediately before reading the post.

4.2 Rendezvous Consistency Window

For the first part of the evaluation, we focus on the cost of enforcing consistency across the application execution.

We start by measuring the *consistency window* [20] for each datastore without using Rendezvous. In our scenario, the consistency window corresponds to the time between the writer function writing the post to the datastore in the EU and the post being read by the reader in the US. Note that, for this metric, in the original application inconsistencies might occur when the post is read, whereas in the Rendezvous version no inconsistencies occur since Rendezvous ensures the post is available before reading. Recall that with Rendezvous, the post is only read after the blocking `WaitRequest` call returns.

The results can be observed in Fig. 4, where, for each datastore, the baseline bars represent the original setup, whereas the above bars concern the setup with Rendezvous. Looking at the figure, we can observe that the additional consistency window imposed by the usage of Rendezvous varies a lot depending on the post's storage, which is a direct consequence of the duration of the `WaitRequest` call. This highlights that the consistency windows are tightly coupled with the consistency mechanisms of each datastore. For instance, Redis requires only an extra 500ms, while S3 takes significantly longer. Most of the increase in the consistency window stems from the latency of replication specific to each datastore.

To confirm this and provide insight into the overhead of just adding Rendezvous, we also measured the time required to write the post in both setups. Fig. 5 presents the time spent writing Rendezvous metadata into the write post operation.

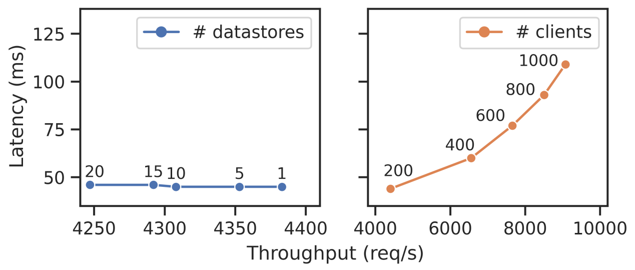

Figure 6. Relationship between throughput and latency observed in a Rendezvous metadata server by varying the number of (i) datastores for a fixed number of 200 clients, and (ii) clients for a fixed number of 1 datastore.

Overall, this additional time never exceeds 70ms and is on average 7% for the 4 datastores. This suggests that the values are minimal in regard to the cost of enforcing consistency, as shown in Fig. 4.

4.3 Rendezvous Scalability

We now focus our evaluation on measuring the performance and scalability of our metadata server. We ran a set of 200 threads per client program, and each machine (1 to 5) was deployed on an AWS EC2 `t2.large` instance with 2 vCPUs and 8 GiB RAM, in the same region as the server (EU). Each individual client registered new branches in the metadata server for a period of 2 minutes.

As depicted in Fig. 6, we conducted two experiments. First, we evaluated the performance by varying the number of datastores written to by a single function using a fixed number of 200 concurrent clients. Then, we changed the number of clients with a single fixed datastore write operation.

For the first experiment, we can observe that the throughput remains approximately the same when ranging from 1 to 20 datastores. Although the throughput is lower for a higher number of datastores, since the metadata increases proportionally with this number, Rendezvous still maintains the same latency of response. This is an adequate performance as we do not expect functions to employ a number of datastores significantly larger than 20.

In the second experiment, we can observe that the throughput increases with the number of concurrent clients, with a peak of approximately 110ms with 1000 clients issuing close to 9000 requests per second.

In comparison with the first experiment, we can infer that the number of clients has a greater impact on the latency than datastores. Overall, we argue that Rendezvous is able to accommodate a large number of clients and their datastores, with reasonable performance and scalability.

5 Related Work

Existing solutions provide communication alternatives between serverless functions, as well as coordination mechanisms for stateful workflows.

Communication in Serverless Applications. Ongoing research already concerns efficient communication and intermediate data sharing in serverless platforms [11, 17, 27, 30]. Both FMI [11] and Boxer [30] offer high-performance and direct communication. FMI provides a messaging interface for communication with collective operations. Boxer demonstrates greater performance with function-to-function communication when compared to resorting to storage systems in data analytics [25]. Alternatively, Pocket [17] uses an intermediate and elastic system to share ephemeral data [16] between functions.

Coordination in Serverless Applications. Current cloud providers offer orchestration techniques [7, 15, 22, 23] that are able to coordinate functions and build stateful workflows. Decentralized and application-level orchestrators [19] may also provide similar guarantees and workflow patterns with existing cloud storage services. However, these solutions do not address the consistency semantics of cloud storage systems and are prone to inconsistencies across regions.

Cross-Service Consistency. Antipode [20] is a decentralized solution that aims to enforce cross-service causal consistency in distributed applications, mitigating consistency violations as those reported in [1]. The work relies on metadata propagation throughout the request execution to capture dependencies and define causality between distinct events. In a similar way to Rendezvous, it can enforce synchronization in actions dependent on replicated information. FlightTracker [28] is a solution that focuses on enforcing stronger session guarantees at Facebook using a centralized server to store clients' metadata. Using a similar centralized approach, Rendezvous also leverages metadata but, in this case, to coordinate serverless applications.

6 Future Work

Rendezvous still exhibits certain limitations, which we aim to address in future research.

Metadata Server Until now, we focused on providing consistency guarantees for serverless functions that share data in geo-replicated datastores, and addressing any consistency violation in read operations. We are aware that Rendezvous currently lacks fault-tolerance guarantees for metadata servers, particularly within the regions hosting a single node. In future research, we aim to tackle these issues by implementing replication strategies within each region, as well as refining the current replication model across regions.

Datastore Monitor Currently, neither metadata server nor datastore monitor support full recovery from a potential datastore monitor crash. Moving forward, we intend to address this in addition to replicating the datastore monitors.

7 Conclusion

In serverless computing, developers frequently turn to geo-replicated storage services to exchange state between functions, which may cause inconsistent executions due to the inherent latency of replication. We proposed Rendezvous, a framework that enhances stateless functions using replicated datastores, allowing for the synchronization of replicated data shared across distinct executions. Rendezvous eliminates inconsistent read operations while preserving the consistency semantics of datastores, and is designed to be integrated within cloud platforms by providing complete transparency to the developers.

Acknowledgments

We express our gratitude to all the anonymous reviewers whose feedback and insightful comments were fundamental to the refinement of this paper. This research has received funding from the Fundação para a Ciência e Tecnologia under the grant UIDB/50021/2020.

References

[1] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. 2015. Challenges to Adopting Stronger Consistency at Scale. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. https://www.usenix.org/conference/hotos15/workshop-program/presentation/ajoux (§5).

[2] Amazon Web Services. 2023. Amazon DynamoDB. https://aws.amazon.com/dynamodb/ (visited on 09/25/2023). (§1, 2, and 3.2).

[3] Amazon Web Services. 2023. Amazon ElastiCache. https://aws.amazon.com/elasticache/ (visited on 09/25/2023). (§1 and 2).

[4] Amazon Web Services. 2023. Amazon Lambda. https://aws.amazon.com/lambda/ (visited on 09/25/2023). (§1).

[5] Amazon Web Services. 2023. Amazon S3. https://aws.amazon.com/s3/ (visited on 09/25/2023). (§1, 2, and 3.2).

[6] Amazon Web Services. 2023. Amazon SNS. https://aws.amazon.com/sns/ (visited on 09/25/2023). (§1).

[7] Amazon Web Services. 2023. AWS Step Functions. https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html (visited on 09/25/2023). (§5).

[8] Peter Bailis and Ali Ghodsi. 2013. Eventual Consistency Today: Limitations, Extensions, and Beyond: How Can Applications Be Built on Eventually Consistent Infrastructure given No Guarantee of Safety? *Queue* 11, 3 (2013), 20–32. https://doi.org/10.1145/2460276.2462076 (§2).

[9] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. https://doi.org/10.1145/2463676.2465279 (§3.2).

[10] David Bermbach and Stefan Tai. 2011. Eventual Consistency: How Soon is Eventual? An Evaluation of Amazon S3's Consistency Behavior. In *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing (MW4SOC '11)*. Article 1, 6 pages. https://doi.org/10.1145/2093185.2093186 (§1 and 2).

[11] Marcin Copik, Roman Böhringer, Alexandru Calotoiu, and Torsten Hoefler. 2023. FMI: Fast and Cheap Message Passing for Serverless Functions. In *Proceedings of the 37th International Conference on Supercomputing (ICS '23)*. https://doi.org/10.1145/3577193.3593718 (§1 and 5).

[12] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*. https://doi.org/10.1145/1294261.1294281 (§1 and 2).

[13] Google Cloud. 2023. Cloud Functions. https://cloud.google.com/functions (visited on 09/25/2023). (§1).

[14] Google Cloud. 2023. Cloud Pub/Sub. https://cloud.google.com/pubsub (visited on 09/25/2023). (§1 and 2).

[15] Google Cloud. 2023. Workflows. https://cloud.google.com/workflows (visited on 09/25/2023). (§5).

[16] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. 2018. Understanding Ephemeral Storage for Serverless Analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. https://www.usenix.org/conference/atc18/presentation/klimovic-serverless (§5).

[17] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. https://www.usenix.org/conference/osdi18/presentation/klimovic (§1 and 5).

[18] Yongkang Li, Yanying Lin, Yang Wang, Kejiang Ye, and Chengzhong Xu. 2023. Serverless Computing: State-of-the-Art, Challenges and Opportunities. *IEEE Transactions on Services Computing* 16, 2 (2023), 1522–1539. https://doi.org/10.1109/TSC.2022.3166553 (§1).

[19] David H. Liu, Amit Levy, Shadi Noghabi, and Sebastian Burckhardt. 2023. Doing More with Less: Orchestrating Serverless Applications without an Orchestrator. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. https://www.usenix.org/conference/nsdi23/presentation/liu-david (§5).

[20] João Ferreira Loff, Daniel Porto, João Garcia, Jonathan Mace, and Rodrigo Rodrigues. 2023. Antipode: Enforcing Cross-Service Causal Consistency in Distributed Applications. In *Proceedings of ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP '23)*. https://doi.org/10.1145/3600006.3613176 (§1, 1, 1, 3.2, 4.1, 4.2, and 5).

[21] Microsoft Azure. 2023. Azure Functions Overview. https://learn.microsoft.com/azure/azure-functions/functions-overview (visited on 09/25/2023). (§1).

[22] Microsoft Azure. 2023. Durable orchestrations. https://learn.microsoft.com/azure/azure-functions/durable/durable-functions-orchestrations (visited on 09/25/2023). (§5).

[23] Microsoft Azure. 2023. What are Durable Functions? https://learn.microsoft.com/azure/azure-functions/durable/durable-functions-overview (visited on 09/25/2023). (§5).

[24] MongoDB. 2023. Change Streams. https://www.mongodb.com/docs/manual/changeStreams/ (visited on 09/25/2023). (§1 and 3.2).

[25] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. https://doi.org/10.1145/3318464.3389758 (§2 and 5).

[26] Olivia Perreault. 2014. Fan had ticket revoked after ticketmaster "Double sold" his floor seat. https://www.ticketnews.com/2018/03/paul-simon-fan-scored-floor-seats-had-them-revoked-by-ticketmaster-after-seat-was/ (visited on 09/27/2023). (§1).

[27] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. https://www.usenix.org/conference/nsdi19/presentation/pu (§1 and 5).

[28] Xiao Shi, Scott Pruett, Kevin Doherty, Jinyu Han, Dmitri Petrov, Jim Carrig, John Hugg, and Nathan Bronson. 2020. FlightTracker: Consistency across Read-Optimized Online Stores at Facebook. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. https://www.usenix.org/conference/osdi20/presentation/shi (§5).

[29] Werner Vogels. 2008. Eventually Consistent: Building Reliable Distributed Systems at a Worldwide Scale Demands Trade-Offs?Between Consistency and Availability. *Queue* 6, 6 (2008), 14–19. https://doi.org/10.1145/1466443.1466448 (§2).

[30] Michal Wawrzoniak, Ingo Müller, Gustavo Alonso, and Rodrigo Bruno. 2021. Boxer: Data Analytics on Network-enabled Serverless Platforms. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. https://doi.org/10.3929/ethz-b-000456492 (§1 and 5).