Toward an Edge-Friendly Distributed Object Store for Serverless Functions

Xin Chen
Georgia Institute of Technology
Atlanta, GA, USA
xchen384@gatech.edu

Manoj Prabhakar Paidiparthy
Virginia Tech
Blacksburg, VA, USA
pmanojprabhakar@vt.edu

Liting Hu*
University of California, Santa Cruz
Santa Cruz, CA, USA
liting@ucsc.edu

ABSTRACT

Serverless computing is changing the way in which we structure and deploy computations in Internet-scale edge systems. This paper presents Capybara, a new scalable and programmable distributed object store for storing and sharing serverless function data objects (state) on edge infrastructures. The key innovations here are (1) achieving scalability and avoiding the significant DRAM cost through a consistent DHT-based P2P architecture; and (2) providing a programmable handler abstraction to customize state management policies (e.g., data caching policies, container "keep-alive" times, access control methods, and data replication policies). We implement Capybara prototype on the Pastry DHT, deploy it on 150 Amazon EC2 nodes, and evaluate it by building several use cases to conduct real-world experiments, demonstrating its significant gains in data locality, state management customization, and scalability compared to the state-of-the-art.

CCS CONCEPTS

• **Computer systems organization** → **Distributed architectures**; • **Information systems** → **Database management system engines**.

KEYWORDS

Distributed object store, serverless functions, edge computing.

*Corresponding author: Liting Hu, Computer Science and Engineering, University of California, Santa Cruz.

APSys '24, September 4–5, 2024, Kyoto, Japan

ACM ISBN 979-8-4007-1105-3/24/09.
https://doi.org/10.1145/3678015.3680485

ACM Reference Format:
Xin Chen, Manoj Prabhakar Paidiparthy, and Liting Hu. 2024. Toward an Edge-Friendly Distributed Object Store for Serverless Functions. In *15th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '24), September 4–5, 2024, Kyoto, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3678015.3680485

1 INTRODUCTION

With the proliferation of 5G and beyond, enabling the next-generation technologies such as smart cities, self-driving cars, online video gaming, and augmented reality require us to ***reconsider*** the way we characterize and deploy these services. Serverless computing is an emerging paradigm, referring to a software architecture where an application is decomposed into 'triggers' (events) and 'actions' (functions), and there is a platform that provides a seamless hosting and execution environment, making it easy to develop, manage, scale, and operate them.

Serverless computing is changing the way in which we structure and deploy computations in Internet-scale edge systems. Many major cloud providers have emerged and introduced serverless computing platforms, including AWS Lambda [7], Google Cloud Functions [15], Microsoft Azure Functions [9], and Apache OpenWhisk [6]. While originally designed for the cloud, the benefits of the serverless paradigm are also vital in Edge/Fog computing environments.

What is edge serverless? To put it simply, edge systems consist of Things, Gateways, and the Cloud. Things are sensors (e.g., smart wearables, car sensors) that "read" from the world and actuators that "write" to it, and Gateways orchestrate Things and bridge them with the Cloud. Figure 1 shows the comparison of "cloud serverless" and "edge serverless". In cloud-based serverless frameworks, functions are executed in containers [12] that are hosted in a cloud datacenter (Figure 1, left). Instead of relying on the cloud to process sensor data and trigger actuators, "edge serverless" decomposes edge applications into serverless functions and executes them in containers on distributed edge nodes (Figure 1, right), i.e., a collection of routers, gateways and micro-datacenter servers maintained by edge providers.

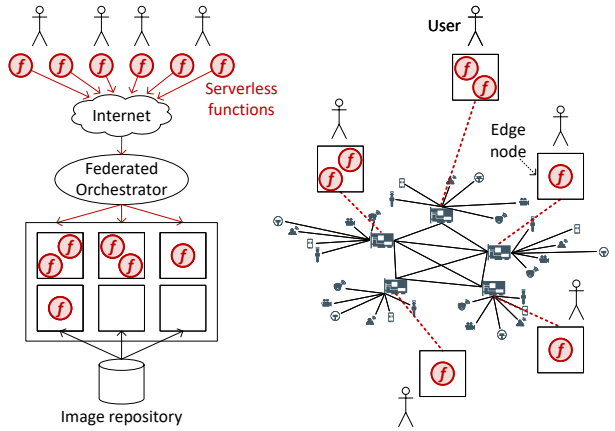


Figure 1: Serverless computing on cloud (left) and edge (right).

Edge serverless can be a game-changer for many latency-sensitive edge applications such as wearable cognitive assistance [29, 32], drone navigation [25], AR-assisted driving [26], and camera networks for surveillance [35]. For example, in an AR gaming platform, a player's interaction with the virtual environment might trigger a real-time object recognition, personalized content delivery, dynamic game element generation, and player analytics, all of which could be handled by separate serverless functions running on edge servers. This serverless edge architecture ensures low-latency responses, enhancing the immersive and responsive nature of the AR gaming experience.

However, existing serverless platforms [7, 9, 14, 15, 18, 19] mostly rely on cloud storage services to store serverless function data, such as AWS S3 [4] and Google Cloud Storage [16]. These solutions, however, are not well-suited for serverless edge applications for the following reasons:

(1) They may cause long delays and strain the backhaul network bandwidth. This is especially the case when functions are instantiated for the first time, container images need to be pulled from remote repository pools (e.g., Docker Hub [13], Amazon ECR [3]) for cold starts, causing expensive data shipping costs and high startup latency.

(2) Additionally, during execution, serverless functions must write their intermediate results to the cloud. For example, a hash-based shuffle from 10^5 *map* tasks to 10^5 *reduce* tasks leads to 10 billion intermediate files being created instantly on the storage system, which may lead significant slowdown due to the lack of local storage.

(3) Offloading security-sensitive data to third-party cloud providers may raise privacy issues.

In this paper, our goal is to build a new scalable and programmable distributed object store for serverless edge applications, enabling millions of edge nodes to be seamlessly integrated as a serverless storage infrastructure.

The challanges. Our work addresses significant challenges due to high diversity and scalability requirements introduced by emerging serverless edge applications.

First, *how to design a distributed object store for storing and sharing function data (state) that supports application-specific customization?* The cloud storage systems, such as S3 [4], Google Cloud Storage [16], and DynamoDB [2], use a client-server architecture, which rely on a central controller or proxy server to manage client requests and data distribution among storage nodes. Unfortunately, there is a critical lack of application-specific customization for state management. Serverless applications and their functions are treated uniformly, with fixed data caching policies, keep-alive times, access control methods, and replication policies. This is problematic for edge applications because they are quite diverse in terms of popularity, invocation frequency, SLO, and state management, necessitating a customizable object store.

Second, *how to scale gracefully to manage a massive number of edge applications' state on millions of edge nodes?* Edge computing presents a unique challenge: clients are distributed geographically, leading to unpredictable workload surges in arbitrary locations. Similarly, edge nodes are geographically distributed, experiencing hardware heterogeneity and churns as they can freely join or leave the systems. This unpredictability makes it exceptionally challenging to implement effective scaling solutions for the object store.

Our solution. To address the above challenges, we present Capybara, a new edge-friendly distributed object store that achieves the desired properties for the serverless edge architecture: *full scalability* and *state management customization.*

The key innovations of Capybara include: (1) It achieves full scalability through a consistent Distributed Hash Table (DHT)-based Peer-to-Peer (P2P) architecture, which avoids the significant DRAM cost of the directory server or metadata server for object lookup or retrieval. (2) It supports state management customization through a programmable handler abstraction. Each serverless application has a set of handlers, allowing users to define procedural code that is executed in response to storage operations, such as *read* and *write*. By doing that, users can customize their own state management policies (e.g., container "keep-alive" times, access control methods, data updating policies, and state replication policies).

In summary, our contributions are as follows:

- We study the software architecture of existing serverless storage systems and discuss their limitations when storing

Storage System	Storage Type	Lookup Unit	DRAM Cost	Read/Write Latency	Co-locate Function with Data	State Management Customization
Amazon S3 [4]	Object storage	Directory	High	High	✗	✗
Google Cloud Storage [16]	Object storage	Directory	High	High	✗	✗
Microsoft Azure Blob Storage [8]	Object storage	Directory	High	High	✗	✗
IBM Cloud Object Storage [17]	Object storage	Directory	High	High	✗	✗
Alibaba Cloud Object Storage [1]	Object storage	Directory	High	High	✗	✗
IndexFS [30]	File system	Directory	High	Medium	✓	✗
InfiniFS [27]	File system	Directory	High	Medium	✓	✗
OpenStack Swift [21]	Object storage	Local index (hashing)	Low	Medium	✗	✗
CRUSH (Ceph) [10, 34]	Object, block, and file	Local index (hashing)	Low	Medium	✗	✗
MapX [33]	Object storage	Local index (hashing)	Low	Medium	✗	✗
CouchDB [5]	Object storage	Local index (B-tree)	Low	Medium	✗	✗
Capybara (this work)	Object storage	Distributed hashing	Low	Medium	✓	✓

Table 1: Comparison of state-of-the-art cloud storage systems and Capybara.

and sharing serverless function data objects (state) in the edge setting at scale.

- We introduce Capybara, a new scalable and programmable distributed object store. To our knowledge, Capybara is the first endeavor to provide an edge-friendly distributed object store for serverless edge applications.
- We implement Capybara on Pastry DHT [31], deploy it on 150 Amazon EC2 nodes, and evaluate it by performing real-world experiments, demonstrating its significant gains in data locality, state management customization, and scalability compared to the state-of-the-art.

2 MOTIVATION AND BACKGROUND

Serverless edge applications generate two types of data objects (state): ephemeral and durable. Ephemeral state is temporary and exists only during the lifetime of an application. For example, for data analytics applications like Spark Streaming [22], their ephemeral state refers to the intermediate results between stages. Durable state, on the other hand, needs to be stored long-term. Examples of the durable state include function's container image metadata, user data, database records, as well as input and output files.

2.1 State-of-the-art solutions

Existing serverless platforms mostly use cloud storage services to store serverless application's data objects (state). In such a system, each data object is uniquely identified by a bit string, called an identifier (Id), name, or key. To manage objects at a massive scale, there are two typical object placement and lookup strategies.

1. Directory-based approach. As shown in Figure 2(a), this approach stores ID-location mappings in a central directory server or metadata server. Clients receive object locations by querying the server. It has the following limitations for storing serverless application's state: (1) *Centralized bottleneck.*

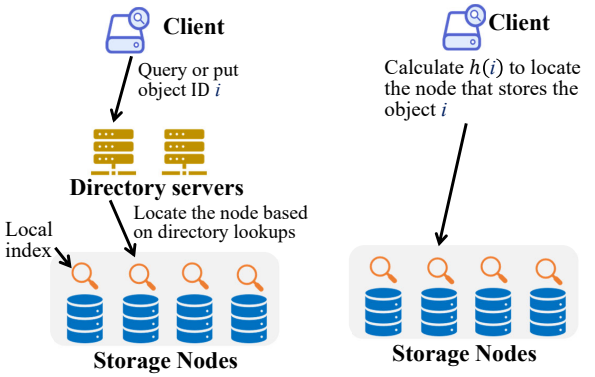


Figure 2: Comparison of object placement and lookup strategies.

The number of function invocations can be quite high, reaching millions per day. The central directory server becomes a bottleneck for handling a large number of requests from clients. (2) *High DRAM cost.* The DRAM resources required to house the directory are significant. For instance, storing 10 billion ID-location mappings requires $> 400GB$, where the majority is used to store IDs, as in practice, the average size of IDs is tens of bytes such as 16 bytes in Ceph [10] and 40 bytes in Twitter [36] or Facebook [23]. This is particularly the case for serverless applications in which the state/ID space ratio is low.

2. Local index approach. As shown in Figure 2(b), this approach places data to storage nodes based on the hash value of its ID $h(ID)$ [24, 34]. Each node in the cluster is responsible for a specific range of data based on the key of the data. Local index approach avoids the overhead of a directory server but may introduce load imbalance, place replicas into the same failure domain, and repeatedly force data re-location when nodes join or leave the system.

2.2 Limitations

We show a comparison among state-of-the-art cloud storage systems in Table 1. Unfortunately, these systems are not suitable for serverless edge applications.

1. Limited scalability. They leverage a client-server architecture, which limits scalability, as they rely on a central controller or proxy server to manage client requests and data distribution among storage nodes.

2. Lack of application-specific customization. These systems do not offer features or interfaces that are tailored to a specific use case, making it difficult to meet different serverless applications' needs. For example, they use a fixed "keep-alive" policy that stores a function's durable state (e.g., container image) in memory after the function execution (the timeout is 10 to 20 minutes), but do not consider application and function's skewed popularity distributions. They use a fixed isolation mechanism, but do not provide different access control methods for different priority data objects.

3 DESIGN

We are not attempting to build a general-purpose distributed object store, such as S3 [4]; such a system would be inappropriate for our needs. Rather, our goal is to support relatively simple operations on data objects (state). Interestingly, even with simple operations, we can build a powerful distributed object store to realize diverse state management policies.

3.1 Architecture

Figure 3 shows the architecture of Capybara. It consists of three components: *DHT-based P2P overlay*, *locality-aware key-value store*, and *programmable handlers*.

Layer 1: DHT-based P2P overlay. All distributed edge nodes are self-organized into a consistent DHT-based P2P overlay, which implements the object-to-node mapping.

Layer 2: locality-aware key-value store. Built upon Layer 1, we implement a persistent storage utility for storing and sharing function data objects (state). Each function has a unique key, which is computed as the secure hash (SHA-1) of the function's name, the application's name, and a randomly chosen salt. Its data objects (state) are stored in the m nodes whose NodeIds are numerically closest to the key. To retrieve a function's state, the routing substrate typically applies a hash function to the key to compute the IDs of the node that store the associated value.

Layer 3: programmable handler abstraction. Unlike other storage systems that only store key-value pairs, we store key-value pairs together with operational code. This operational code is structured as a set of *programmable handlers* (e.g.,

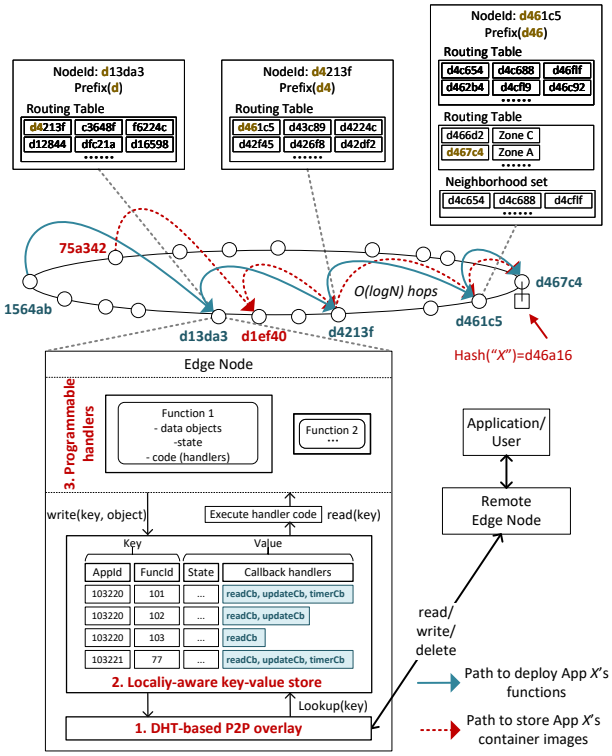


Figure 3: The Capybara system architecture.

`readCb`, `updateCb`, `timerCb`) that specify how the application behaves, for example, how it modifies state when certain events occur. It can make dynamic decisions based on its access history, its current number of replicas, and the time of day. For example, when a client performs a "*read*" operation to access a data object (e.g., a function's container image), the `readCb` handler will be invoked to perform a simple operation, such as incrementing a counter for the number of *reads*. If the counter increases rapidly in a short period of time, the container image will be considered "popular". Then the handler can dynamically change the keep-alive time of the container image to make it retain longer in memory to reduce cold-start latency.

The key to efficiency comes from several factors. 1. *Scalability and low DRAM cost.* We use a decentralized architecture, allowing data to be stored and retrieved directly between nodes without the need for a central directory (metadata) server. Therefore, it avoids the significant DRAM cost and can scale horizontally, as more nodes can be added to the network as needed. 2. *Application-specific customization.* We abstract away the complexities of DHT-based P2P ring overlay construction, routing substrate for data placement and lookup. Our system can easily support various storage lifetimes, access control methods, or state replication policies.

Programmable Handler	Description
`createCb(caller)`	Invoked upon the initial creation of a function's state (data object), such as during application registration. Returns the state to be stored by the node (e.g., itself or nil).
`readCb(caller, args)`	Invoked when a *read* operation is performed on the state (e.g., when serverless scheduler tries to access function's container info during invocation). Returns the function state. It may modify the state and write it back to the storage system depending on the handler code.
`updateCb(caller, new_state)`	Invoked when updating an existing function state (data object). Returns the new state that needs to be stored.
`timerCb()`	Invoked periodically at intervals set by the edge zone administrator. This handler has no return value. It is used to perform periodic tasks such as container cleanup, state replication, and health monitoring.

Table 2: Capybara programmable handlers (we use state and data object interchangeably in this paper).

3.2 DHT-based P2P Overlay

As the first layer, all distributed edge nodes are self-organized into a consistent DHT-based P2P ring overlay, which is similar to the BitTorrent nodes that use the Kademila DHT [28] for "trackerless" torrents. Each edge node is assigned a unique 128-*bit* `NodeId` in a very large circular Id space (e.g., $0 \sim 2^{128}$). NodeIds are used to identify the nodes and route queries for object placement and lookup. DHT-based routing substrate guarantees that, *no matter where the function is invoked, we can find nodes that store its data objects (state) within* $O(logN)$ *hops, where* N *is the total number of nodes in the system.* To do that, each node needs to maintain a routing table. The routing works based on prefix-based matching. Every node knows m other nodes in the ring and the distance of the nodes it knows increases exponentially. It jumps closer and closer to the destination, like a greedy algorithm, within $\lceil log_{2^b} N - 1 \rceil$ hops, where $2^b - 1$ is the number of entries in the routing table.

3.3 Locality-aware Key-Value Store

As the second layer, we create a "bucket" data structure in each edge node and organize them into a locality-aware distributed key-value store. The key innovation is co-locating function data with function invocations on edge nodes that minimize the data shipping cost, leveraging the same DHT overlay networks for placing functions and their data objects in $O(logN)$ steps regardless of their geographical locations.

When an application joins the system, an application certificate is generated, which assigns the application a unique 160-*bit* key (`AppId`), e.g., the secure hash (SHA-1) of the application's textual name, the owner's Id, and a random salt. When inserting data, Capybara routes the data to the k nodes whose NodeIds are numerically closest to the 128 most significant bits of the key (k is a user-defined parameter with a default value of 3). When retrieving data, Capybara applies the same hash function to the key to compute the NodeIds that store the data object.

Capybara deploys an application's functions on edge nodes using the same DHT-based routing substrate. The deployment process includes (1) generating a key by calculating the secure hash of the application's key (`AppId`); (2) routing an "invocation query" toward the key, which specifies function code and triggers; (3) delivering the query to the node whose NodeId is numerically closest to the key, and (4) spawning containers on this node and/or the neighboring nodes. When the workload changes, the neighboring nodes are used for scaling containers.

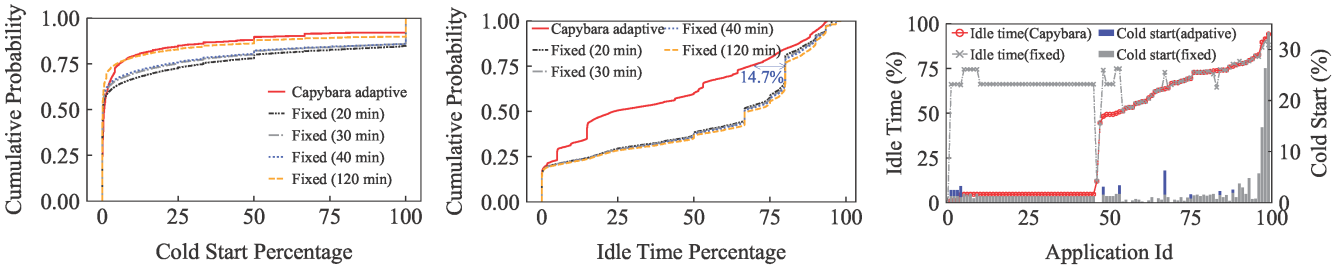
This procedure ensures (1) the data insertion and function deployment share the same key ($key = hash(\text{"}application\text{"})$), so their routing paths converge at the same destination node. Therefore, the function data is placed close to the node where the function will be invoked, enabling data locality. As shown in Figure 3, application X's functions and X's container images are placed around the same location; and (2) following with the load balance property of DHTs, the function data are stored in a well-balanced manner.

3.4 Programmable Handlers

```
var readCb = function (state, args) {
     state.intervals.add(
          sys.currentTime() - state.lastExecTime,
     );
     state.keepAlive =
          dht.currentTime() + max(intervals) * minutes;
     state.coldWait =
          dht.currentTime() + min(intervals) * minutes;
     write(this, sys.getKey(), state);
     return state;
};
var timerCb = function (state) {
     if (sys.currentTime() > state.coldWait) {
          sys.deployContainer();
          reset(state.coldWait);
     }
     if (sys.currentTime() > state.keepAlive) {
          sys.deleteContainer();
          reset(state.keepAlive);
     }
     write(this, sys.getKey(), state);
};
```

Listing 1: Adaptive keep-alive policy.

As the third layer, we develop a new programmable handler abstraction (Table 2). We implement a store controller for



(a) CDF distribution of function instance's cold starts for 15k Applications.

(b) CDF distribution of function instance's idle times for 15k Applications.

(c) Cold start and idle time trade-off for 100 Applications.

Figure 4: Capybara's adaptive keep-alive policy.

executing these handlers. When a request is sent to the key-value store to access the data, it spawns a code runner process to retrieve and execute the relevant handler code.

Listing 1 shows the code snippets of an adaptive keep-alive policy. Here, we introduce two parameters: *cold-waiting window* and *keep-alive window*. The *cold-waiting window* represents the time between the last execution and when the system loads the function image to memory. The *keep-alive window* determines how long the application will remain in memory after the last execution or after its image is loaded to memory. If idle times are consistently short, indicating a high invocation frequency, we will reduce the cold-waiting time, causing the image to be loaded to memory more frequently. The `readCb` handler is invoked when the function state is read. The `timerCb` handler wakes up periodically, which is responsible for loading the function image to memory after the cold-waiting window, recycling the container after the keep-alive window, and resetting both windows.

4 EVALUATION

We performed a feasibility study of Capybara. We resort to a cluster of 150 Amazon EC2 nodes, each of which has 4 vCPUs, 16GB of RAM, and 32 GB of disk space (equivalent to Cisco's IoT gateway [11]). To create a real-world heterogeneous edge environment, we launched 5000 heterogeneous edge nodes (emulated using JVMs) on the testbed. Each node can randomly host up to 4, 16, 64, or 256 functions.

We implement the adaptive keep-alive policy on Capybara for storing severless function's container images, driven with Microsoft Azure traces [20]. The traces include 15,940 Applications that consist of 39,491 functions in total. We compare Capybara's adaptive keep-alive policy with state-of-the-art serverless systems' fixed policies (20 minutes, 30 minutes, 40 minutes, and 120 minutes).

- *Cold Start (%)*: Percentage of Apps resulting in a cold start.
- *Idle Time (%)*: Percentage of time that the spawned application's containers remain idle without any function invocation (measured at 1 minute granularity).

Figure 4a shows the comparison of the percentage of cold start out of the total number of invocations for different policies. Results show that Capybara reduces the cold start percentage at the 90_{th} percentile by 42.9%, as compared to the fixed keep-alive policy (120 minutes). The improvement is more noticeable for shorter fixed keep-alive time (20 minutes, 30 minutes, 40 minutes).

Figure 4b shows the comparison of the percentage of container idle times out of the total number of invocations for different policies. Results show that Capybara reduces the container idle time percentage at the 90_{th} percentile by at least 3.2%~4.3% and 75_{th} percentile by 14.7%, as compared to the fixed keep-alive policies.

Figure 4c shows a trade-off analysis between idle time and cold starts for 100 randomly selected individual applications. Intuitively, if a policy reduces cold starts, it tends to keep container images in memory longer, leading to increased idle time and resource wastage. Therefore, an effective policy should strike a balance between idle time and cold starts. Results show that Capybara's adaptive keep-alive policy has less idle time with comparable cold starts, as compared to a fixed keep-alive policy (20 minutes), and is thus more resource-efficient.

5 CONCLUSION

Capybara is a new scalable and programmable distributed object store designed for serverless edge applications. The preliminary results are quite encouraging and clearly show the potential of our approach. Our future work involves implementing more diverse state management policies and implementing a secure runtime, like a language-based sandbox to prevent handlers from interfering with other applications or consuming excessive resources.

ACKNOWLEDGMENTS

This work is supported by the National Science Foundation (NSF-CAREER-2313737, NSF-OAC-2313738, and NSF-CNS-2322919).

REFERENCES

[1] Alibaba Cloud Object Storage Service (OSS). https://www.alibabacloud.com/product/object-storage-service.

[2] Amazon DynamoDB. https://aws.amazon.com/dynamodb/.

[3] Amazon Elastic Container Registry. https://aws.amazon.com/ecr/.

[4] Amazon S3. https://aws.amazon.com/s3/.

[5] Apache CouchDB. http://couchdb.apache.org/.

[6] Apache OpenWhisk. https://openwhisk.apache.org/.

[7] AWS Lambda. https://aws.amazon.com/lambda/.

[8] Azure Blob Storage. https://azure.microsoft.com/en-us/products/storage/blobs.

[9] Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[10] Ceph. https://docs.ceph.com.

[11] Cisco Kinetic Edge & Fog Processing Module (EFM). https://www.cisco.com/c/dam/en/us/solutions/collateral/internet-of-things/kinetic-datasheet-efm.pdf.

[12] Docker Container. https://www.docker.com/resources/what-container/.

[13] Docker Hub Container Image Library. https://hub.docker.com/.

[14] Function Compute, Alibaba Cloud Function Compute. https://www.alibabacloud.com/product/function-compute.

[15] Google Cloud Functions. https://cloud.google.com/functions.

[16] Google Cloud Storage. https://cloud.google.com/.

[17] IBM Cloud Object Storage. https://www.ibm.com/cloud/object-storage.

[18] Knative. https://knative.dev/.

[19] Kubeless. https://kubeless.io/.

[20] Microsoft Azure Function Traces. https://github.com/Azure/AzurePublicDataset.

[21] OpenStack Swift. https://github.com/openstack/swift.

[22] Spark Streaming. https://spark.apache.org/streaming/.

[23] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[24] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures. In *Proceedings of the 20th International Middleware Conference*, Middleware '19, page 41–54, New York, NY, USA, 2019. Association for Computing Machinery.

[25] Samira Hayat, Roland Jung, Hermann Hellwagner, Christian Bettstetter, Driton Emini, and Dominik Schnieders. Edge computing in 5g for drone navigation: What to offload? *IEEE Robotics and Automation Letters*, 6(2):2571–2578, 2021.

[26] Patrick Lindemann, Tae-Young Lee, and Gerhard Rigoll. Supporting driver situation awareness for autonomous urban driving with an augmented-reality windshield display. In *2018 IEEE International Symposium on Mixed and Augmented Reality Adjunct (ISMAR-Adjunct)*, pages 358–363, 2018.

[27] Wenhao Lv, Youyou Lu, Yiming Zhang, Peile Duan, and Jiwu Shu. InfiniFS: An Efficient Metadata Service for Large-Scale Distributed Filesystems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 313–328, Santa Clara, CA, February 2022. USENIX Association.

[28] Petar Maymounkov and David Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, page 53–65, Berlin, Heidelberg, 2002. Springer-Verlag.

[29] Manuel Olguín Muñoz, Roberta Klatzky, Junjue Wang, Padmanabhan Pillai, Mahadev Satyanarayanan, and James Gross. Impact of delayed response on wearable cognitive assistance. *Plos one*, 16(3):e0248690, 2021.

[30] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, page 237–248. IEEE Press, 2014.

[31] Antony I T Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350, London, UK, UK, 2001. Springer-Verlag.

[32] Mahadev Satyanarayanan and Nigel Davies. Augmenting cognition through edge computing. *Computer*, 52(7):37–46, 2019.

[33] Li Wang, Yiming Zhang, Jiawei Xu, and Guangtao Xue. MAPX: Controlled Data Migration in the Expansion of Decentralized Object-Based Storage Systems. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 1–11, Santa Clara, CA, February 2020. USENIX Association.

[34] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, pages 31–31, 2006.

[35] Zhuangdi Xu, Harshil S Shah, and Umakishore Ramachandran. Coral-pie: A geo-distributed edge-compute solution for space-time vehicle tracking. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, page 400–414, New York, NY, USA, 2020. Association for Computing Machinery.

[36] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208. USENIX Association, November 2020.