Cliffhanger: An Experimental Evaluation of Stateful Serverless at the Edge

Adam Hasselberg
RISE/KTH
adam.hasselberg@ri.se

Thomas Ohlson Timoudas
RISE
thomas.ohlson.timoudas@ri.se

Paris Carbone
KTH
parisc@kth.se

György Dán
KTH
gyuri@kth.se

***Abstract*—The serverless computing paradigm has transformed cloud service deployment by enabling automatic scaling of resources in response to varying demand. Building on this, stateful serverless computing introduces critical capabilities for data management, fault tolerance, and consistency, which are particularly relevant in the context of distributed deployments, notably in edge computing environments. In this work, we explore the feasibility of stateful serverless computing in resource-limited edge environments through an empirical study utilizing a multi-view object tracking application. Our results show that while these systems perform well in cloud environments, their effectiveness is severely affected at the edge due to state, application, and resource management solutions optimized for cloud environments. Existing solutions are most detrimental to applications with intermittent workloads, as typical combinations of concurrency handling and resource reservation can lead to minutes of unstable system behavior due to cold starts. Our results highlight the need for a tailored approach in stateful serverless systems for edge computing scenarios.**

***Index Terms*—Distributed computing, Edge Computing, Fog computing**

I. Introduction

Serverless computing emerged out of cloud computing in the last decade, offering a programming interface to cloud users that replaces virtual machines with short-lived function instances, and is often referred to as Function-as-a-Service (FaaS). Serverless platforms today provide tools to compose cloud applications out of FaaS-deployed functions. The two major advantages of serverless computing include autoscaling to fluctuating workloads and a pay-as-you-go pricing model, i.e., a pricing strategy purely based on usage rather than uptime. This eliminates the need for pre-allocated compute capacity or dedicated infrastructure and may lead to significant cost savings [1]. Recently, serverless platforms expanded their capabilities to manage application state natively, in addition to compute resources, in the serverless model [2], hence, automating scalability, isolation, and fault-tolerance when managing applications that use persistent state. This has greatly expanded the usability and potential of employing such systems to build general-purpose complex and distributed applications.

Flexibility, scalability and fault tolerance make stateful serverless computing a promising service abstraction for edge computing deployments, where workloads are often dynamic and require reliable and timely processing. Existing studies on stateful serverless at the edge have primarily focused on the geographical distribution of state and computations [3], [4], and rely either on analytical models, simulations, or modified solutions for state management that are made locality and workload aware [5]. It is, however, unclear how cloud-native serverless frameworks perform under resource constraints, including limited memory, storage, and computing capacity, typical of edge deployments.

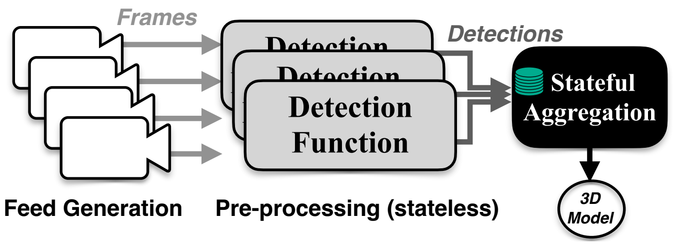

Fig. 1. An overview of MVOT application

Motivating example: As a representative example of stateful serverless at the edge, we consider multi-view 3D object tracking (MVOT). The use case is of high relevance to today's emerging requirements in wireless on-demand services and addresses modern needs in manufacturing, smart infrastructure, and smart cities, with the latter being piloted in large cities for monitoring, security, and traffic management [6].

In MVOT multiple cameras capture parts of a scene from different angles and the objective is to produce a live 3D model of moving objects (e.g., pedestrians or cars) across all input video feeds. Computation must keep up with the frame rate of the cameras, and, achieving low end-to-end latency is crucial for applications consuming the 3D model. Figure 1 illustrates how such an application may be composed using two functions, one stateless parallelizable task, and one stateful aggregation. The application may benefit from *scale out* over compute resources since different camera feeds can be pre-processed independently of each other and later be aggregated together. Conversely, during periods of no activity, the application is expected to *scale down* or halt its operation and release edge resources, such that they may be used by other applications. Aggregating the live feed for tracking is a stateful computation, i.e., at each aggregation step, the preceding computational state must be used together with the set of new data. The state of the aggregation should be managed and stored reliably, allowing the computation to *migrate* seamlessly across available physical resources and the system to maintain optimal performance and resource utilization. The MVOT application is representative of various latency-sensitive sensor data applications with the following characteristics: independent pre-processing of events using ML inference, information fusion, and in-order aggregation.

Ideally, an edge deployment of the application would combine the elasticity and simplicity of common serverless cloud platforms, with low-latency, localized computing at the edge. To that end, we present *Cliffhanger*, an experimental evaluation study to identify performance characteristics showcasing the limitations of existing serverless software in edge deployments. We make use of the MVOT workload as a mini-benchmark and evaluate the performance impact of key configuration parameters related to edge serverless resource management. These include concurrency provisioning (parallelism), reserved compute resources per instance (vCPUs), and state configuration mode (externally vs internally managed state). We assess the capabilities of existing serverless frameworks and discuss challenges, establishing a basis for future research toward realizing performant stateful serverless at the edge.

Our main contributions are as follows:

- We analyze existing studies in stateful serverless and identify the need for a comprehensive performance evaluation of existing cloud serverless software at the service of edge application needs, with a focus on constrained resources (section II).
- We identify a set of key configuration parameters and deployment options for stateful serverless at the edge and construct a set of experiments based on a representative edge application, focusing on end-to-end latency and resource utilization (section III).
- We present a thorough analysis of the results revealing the importance of previously overlooked aspects related to local optimizations for stateful edge serverless such as resource sharing, rate limiting, and batching (section IV).
- We discuss the implications of the results with regards to future stateful serverless systems for the edge, and in relation to related work in the wider context of serverless computing (section V).

II. Background

A. System Architectures for Stateful Serverless

We consider stateful serverless as a combination of three components: application orchestration, state management, and compute management. The application layer deals with I/O to external systems, and orchestrations of functions (also called *workflows*). The state layer manages the persistence and consistency of the state. The compute layer is where the actual physical computation occurs. In any stateful serverless system delivering strong guarantees on the semantics of state updates and function invocations (e.g. exactly-once processing) state- and application- management are intimately intertwined.

Existing stateful serverless systems can be broadly divided into two categories: (1) On-top-of serverless, and (2) Self-contained. On-top-of serverless systems (e.g., *Flink Statefun* [7] and *Kappa* [8]) orchestrate stateful applications on top of serverless systems, effectively only managing the State and Application-level of the deployment, while relying on an independent deployment of a FaaS-system to manage the Compute resources. Self-contained stateful serverless systems (e.g. *Durable Functions* [9] and *Cloudburst* [10]) manage all three layers, enabling joint optimizations and physical colocation of state and compute, at the cost of greater system complexity.

B. Related Work

Surveying existing systems: Raith, Nastic and Dustdar [11] conducted a survey of 45 existing frameworks for serverless at the edge from both industry and academia. The survey provides a comprehensive overview of the state of serverless edge computing frameworks and assesses the maturity levels, revealing that many current solutions lack sophistication in terms of performance and infrastructure optimization.

Workload placement at the edge: Cicconetti, Conti, and Passarella [4] explore the topic of stateful serverless at the edge, evaluating different state management strategies. They show that by keeping the state local and, more importantly, minimizing state propagation over the network, stateful serverless edge systems can improve application latencies and reduce traffic volumes. On the same note, Xu et al. [3] present efficient algorithms for optimal serverless application placements in edge computing, their final algorithm uses online learning which performs application placements reactively. Through simulations comparing their algorithm to naive counterparts, they find that they can reduce the total cost by 32% and achieve a 27% average reduction in latency by optimizing function placements on distributed resources.

Resource utilization in the cloud: While the aforementioned workload placement strategies demonstrate significant improvements in serverless applications, yielding up to a 32% reduction in total cost and a 27% average reduction in latency, these approaches predominantly focus on high-level optimizations for geographically distributed resources and workloads. In contrast, Li et al. [12] show that performance-aware resource-efficient scheduling in cloud-serverless systems can reduce costs by 42%. Their solution optimizes CPU utilization per instance by using SLO-aware scheduling on top of an external FaaS system, emphasizing detailed resource management at the server level. This distinction is crucial; while optimization strategies focusing on geographical distribution provide a broad framework for efficiency, the detailed, resource-efficient scheduling techniques by Li et al. delve deeper into the fine-grained aspects of serverless system performance and achieve a 42% reduction in operational costs. This suggests that while geographical distribution strategies lay a solid foundation for efficiency, integrating them with detailed, resource-specific scheduling could further enhance the performance and cost-effectiveness of stateful serverless systems at the edge.

Edge environments, unlike cloud deployments, are inherently resource-constrained, and utilization of the edge resources is crucial. Recognizing this, our research aims to complement existing studies by identifying the implications of applying state-of-the-art cloud-tailored stateful serverless technologies in environments with severe resource limitations. The subsequent section outlines our methodology, which includes a series of experiments designed to rigorously evaluate various stateful serverless configurations and approaches.

III. Evaluation Methodology

In this section, we present five experiments based on a prototype of the motivating example application. The structure of this section is as follows: III-A details how the application is implemented and deployed using a stateful serverless system. III-B describes the five different experiments we conduct using the application. III-C describes the different state and application management approaches we compare, and finally III-D details the different stateful serverless configuration options compared using the experiments.

A. Implementation

Our evaluation scenarios have been built based on the MVOT application described in the introduction. We implemented all functions entailed in multi-view object tracking (pre-processing/detection, aggregation into live 3D view) using a combination of cloud serverless library configurations. We further generated an input workload from a public dataset consisting of 7 camera feeds [13]. In this section, we detail the design choices, configurations and metrics used.

1) Application Description: For an extended description of MVOT application we refer to the original paper [14]. In short, the approach is based on pre-trained object-detection models like YOLO [15] to detect objects in each frame individually. Subsequently, the full set of detections at each time step are aggregated using recursive Bayesian inference.

The approach conveniently maps to two distinct functions, first a highly parallelized stateless pre-processing of each frame using ML inference, followed by a stateful aggregation, conceptually illustrated in Figure 1. At the moment of writing no practical implementations of such an aggregator exist, and the authors report a per-frame processing time of up to 20 seconds for their unoptimized prototype. Attempting to actually deploy a real implementation would immediately restrict the experiment to be a benchmark of the specific Bayesian aggregation implementation. Instead, we replace the Bayesian aggregator with a simple operator which we call *aggregator*. The aggregator in our experiment receives detections, which it stores as its state and once all detections for a timestep have been received it outputs that set of detections, in order, and evicts the set from its state, preventing the state from ever growing too large. In the last experiment we present, we use a modified version of the aggregator which retains all detections indefinitely, causing its state to continuously increase in size.

2) Experiment execution: The experiment uses 2-primary services which together form a stateful serverless system: Flink Statefun (v3.2.0) for state and application orchestration and Knative for serverless resource management (v1.10.0), both deployed on the same server using Kubernetes. We use an experiment-runner written in Python which runs natively on the same host server. The runner reads image files produced from seven video files corresponding to seven cameras, sends the frames as separate events at a given FPS, and records the send time for each frame. The application manager consumes the frames in the order they were written and invokes the detection function. The serverless resource manager receives the detection invocations and dispatches them to an available function instance (or spawns a new instance). The function instance executes the detection function and returns the result to the application manager, which then invokes the aggregator with the new detection as an argument, and includes the current state of the aggregator as a second argument, with each invocation. If any aggregations were completed during the invocation (note that each aggregation requires 7 detections) the result is returned to the application manager, which finally returns the result to the experiment runner.

B. Experiments

We create 5 different experiments, all based on the same object-tracking application. For each experiment, we employ a different subset of configurations focusing on different key aspects of stateful serverless performance at the edge.

1) The **Provisioned Concurrency** experiment establishes stable performance using a static number of pre-provisioned serverless instances, comparing the performance impact of different stateful serverless configuration options when operating near the maximum throughput.
2) The **State and application management** experiment compares different state-management and application-orchestration methods when operating safely below the maximum sustainable throughput, using identical serverless configuration options.
3) The **No Provisioned Concurrency** experiment compares scale-up time using different configuration options.
4) The **Minimum Overhead** experiment uses a modified detection function and workload, causing the system to be throughput-limited by the aggregator function.
5) The **State size** experiment uses a modified aggregator function such that its state grows linearly over time, allowing us to see how different state management approaches are affected by state size.

C. State and Application Management Approaches

Stateful serverless is realized by three main components: Application management, state management and resource management. In all our experiments we use *Knative Serving* for resource management, which receives HTTP requests from the application manager, and scales or load balances the function instances accordingly. Application management ties the functions together, i.e. ensuring that the output of the detection function is forwarded to the aggregator function, and communicating with external systems, i.e. the experiment runner. State management entails where and how the state is managed. In our experiments, we compare a total of 5 different

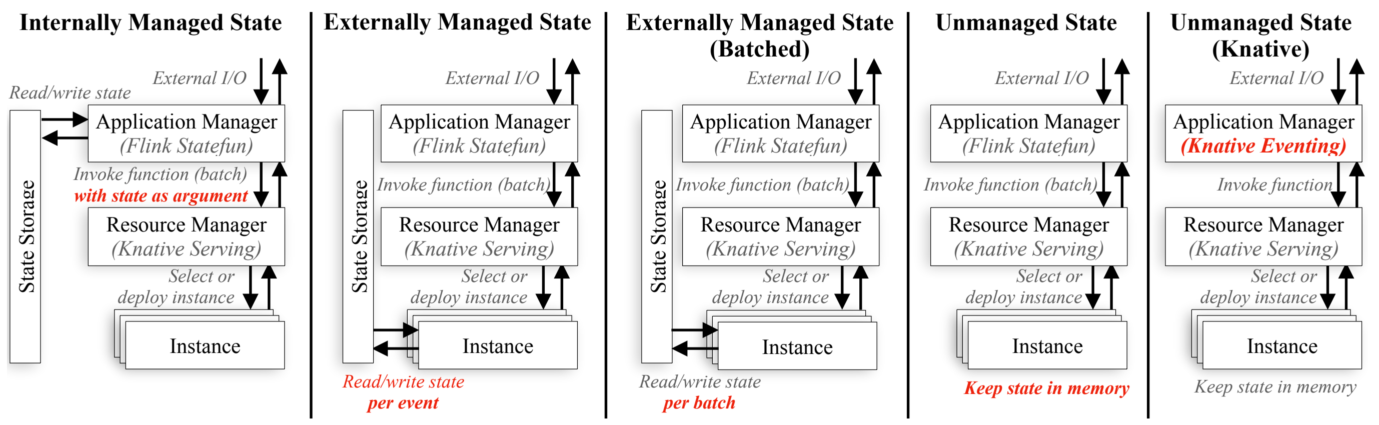

Fig. 2. Five state and application management approaches.

variants realizing stateful serverless, illustrated in Figure 2, from left to right:

1) *Internally managed state* (abbreviated as `Internal` henceforth): The application manager manages both the application and the state.
2) *Externally managed state* (`External`): We move the state out of the application manager and instead let the aggregator read and update its state from external storage[1] each time it processes an event.
3) *Externally managed state (batched)* (`External (batched)`): When the function handler receives a batch of events it immediately reads the state from storage before processing the batch, and writes the final state after processing the entire batch.
4) *Unmanaged state* (`Unmanaged`): We let the state reside entirely within the memory of the single function instance serving the aggregator function, sacrificing fault tolerance and durability.
5) *Unmanaged state (Knative)* (`Knative`): We replace the application manager used in the preceding variants (Flink Statefun) with Knative Eventing, and leave the state unmanaged (no fault tolerance and durability).

D. Serverless Resource Configuration

Along with comparing different application and state management methods, we also consider various combinations of three commonly used resource configuration options for the serverless deployment of the functions. We examine these parameters using only the *internally managed state* variant.

1) Reserved/Unreserved Resources: Cloud providers offering serverless always reserve both vCPU and memory for function instances, forcing the serverless users to select an instance type from a set of predefined instance sizes. However, setting up our own serverless platform using Kubernetes (and Knative) allows us to skip reserving vCPUs per function instance. Note that reserving a vCPU for a function instance implies that the instance has guaranteed access to the reserved amount, but it is also confined to what has been reserved for it. Without reservation, the different function instances share the available processing capacity, and the detection function within each instance may utilize parallelized execution for faster inference. In our experiments, up to 60 instances can reserve 1 vCPU each.

2) Dedicated/Shared Instances: Stateful application managers built on-top-of serverless, such as Flink Statefun, require that the serverless functions are instrumented for the stateful library. For simplicity, we implement both the aggregator and the detection functions within the same Python file and construct the same function image. In the *dedicated instances* configurations, we deploy this identical function image twice, as two different serverless Functions, with separate configurations that are scaled independently of each other.

With *shared instances*, we deploy a single serverless function, with a single configuration. Effectively, all invocations received by the resource manager are identical black-box invocations, and the invocations for the two different functions *share the same queue* outside of the application manager.

3) Minimum/Maximum Concurrency: All FaaS frameworks permit users to define the minimum number of function instances to be active at all times to avoid cold start (sometimes called provisioned concurrency), and the maximum number of function instances, to avoid excessive costs. In our experiments, the aggregator does not use concurrent function instances, while for the detection function (or *shared instances*) we vary the maximum number of instances when using unreserved vCPU.

E. Hardware Resources

The experiments were executed on a HPE ProLiant DL380 Gen10 server equipped with two Intel Xeon Gold 5218 CPUs @ 2.3 GHz with 16 cores each, supporting two threads each for a total of 64 vCPUs. The subsystems we measure are all deployed using Kubernetes MicroK8s v1.26.5 revision 5395 running on a single Ubuntu 20.04.6 LTS server.

[1]We use MinIO for storage, an open-source S3 compatible object store.

ISBN: 978-3-903176-61-4

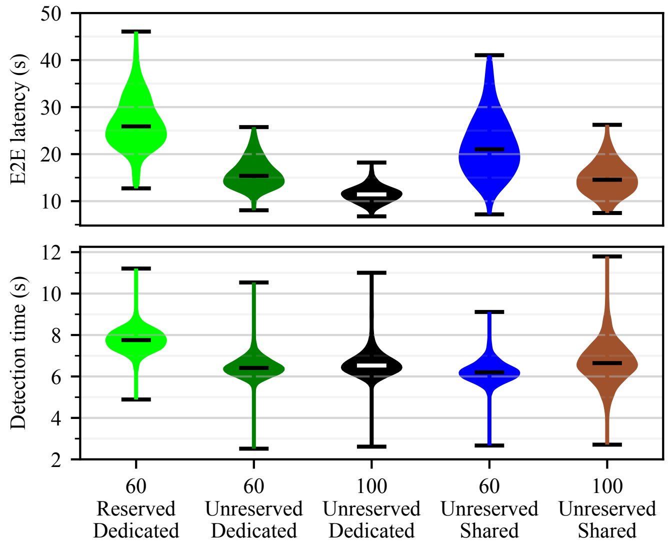

Fig. 3. End-to-end latency (top) and detection time (bottom) for five serverless-configurations, using a workload near the maximum throughput with provisioned concurrency.

IV. Results

In this section, we present the results from each experiment and our findings, one at a time.

A. *Provisioned Concurrency*

Figure 3 shows the end-to-end latency and detection time, using a steady workload of 1 frame per second per camera, which is close to 100% of the maximum throughput. We compare different configurations where all instances are pre-allocated (referred to as provisioned concurrency in some serverless offerings) when the experiment starts. Hence, there is no cold start involved in executing the functions.

Reserved resources interfere with parallel processing: The bottom plot in Figure 3 shows the detection times obtained using different configurations. Detection time is measured within the function instances and only measures the time for the CPU-heavy detection task. The average detection time is around 6-8 seconds for all configurations, and hence it is reasonable to expect that 60 instances would be able to sustain 7 detection tasks per second with 8 seconds per detection task on average.

One may expect that reserving computational resources for instances would be beneficial for achieving low latency, but this is not the case. It is in fact the `60 Reserved Dedicated` configuration that results in the highest E2E latencies. We attribute the comparatively poor performance to the library we use for object detectionwhich can utilize thread parallelism, but it has no benefit from this when the instances are constrained to a single CPU. We also tried using 30 instances with 2 vCPU's reserved each but found that such a configuration was unable to keep up with this workload, resulting in an ever-growing E2E latency.

More concurrency leads to lower latency: The detection time metric only captures the exact time it takes to perform the actual detection computation, it does not capture queueing time. In principle, each instance should have some amount of idle time in between detections, while it is waiting for network I/O. By increasing the number of instances beyond the number of available CPUs to 100 as in the `100 Unreserved Dedicated` configuration, the instances' idle time yields CPU time to other instances, and we achieve even better results. The instances are able to share the CPU time efficiently and result in the lowest E2E latency. These results show that one should plan for more instances than expected based on the workload to achieve optimal performance.

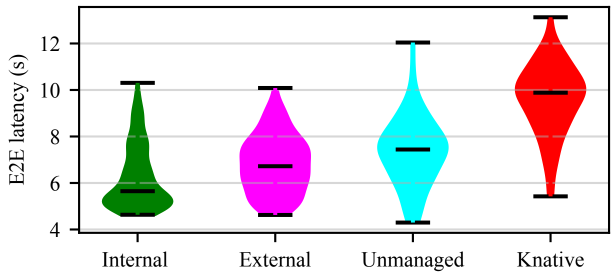

Fig. 4. End-to-end latency and detection time for four system configurations, using a workload safely below the maximum throughput with provisioned concurrency

Rate limiting invocations improves performance: Instance sharing is the arguably easiest way to build applications using Flink Statefun as it only requires building and deploying a single function image, but also leads to the functions sharing the same queue. Unsurprisingly, we find that the `60 Unreserved Shared` configuration results in very high E2E latencies as the events for the aggregator are queued up behind the events for the detection function. However, by increasing the number of shared instances to 100 as in the `100 Unreserved Shared` the latencies decrease. Here we find, through inspection of system logs, that Statefun periodically activates *flow control* and rate limits invocations for the detection functions. The cycles of flow control, halting and then resuming invocations, cause burstier behavior which in-turn produces greater variance in detection times. More importantly, however, flow control maintains a more even pace between the two functions, reducing the overall E2E latency.

B. *State and Application Management*

In Figure 4 we compare state management configurations when the workload is reduced by 25% so that the systems are safely below the maximum capacity. This leads to even lower detection times and E2E latencies. To better compare the state and application management we use the same number of unreserved and pre-provisioned instances for all variants.

Reducing the critical path leads to better performance: Comparing the `Internal` variant to `External` in Figure 4 we observe that the end-to-end latency is greater when the state is stored externally. The external state access occurs for each event the aggregator processes, and it must combine 7 events to complete an aggregation. This result shows that despite the

ISBN: 978-3-903176-61-4

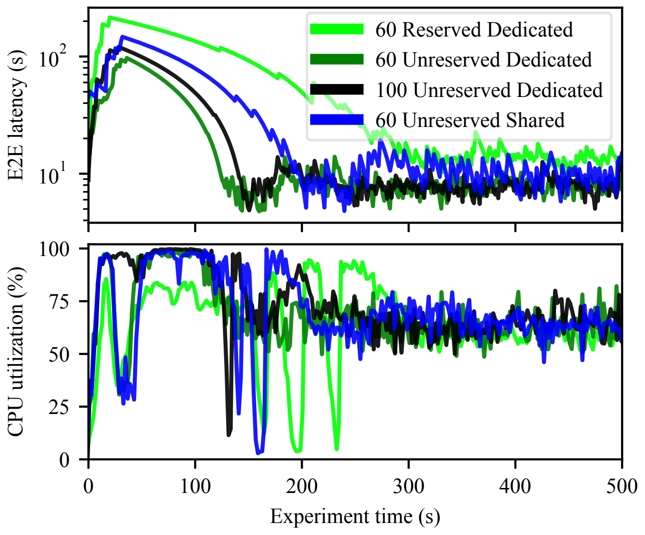

Fig. 5. End-to-end latency vs. time (top), and total host CPU utilization vs. time (bottom) for four configurations using no provisioned concurrency.

network latency being minimal (everything happens on the same physical host), reducing the number of sub-system hops still reduces E2E latencies significantly.

Unmanaged state fails sooner rather than later: Both the `Unmanaged` and `Knative` variants show greater variance and higher E2E latencies. Theoretically, these variants should represent optimal state management as they never transfer or persist the aggregators' state. However, we find that they are too unreliable, despite running the experiment many times the variants consistently fail within the first 150 seconds, which leads to consistently worse results.

Application management performance is important: The `Knative` variant is the only variant we show that does not use Flink Statefun at all. We find that this significantly increases E2E latency as the alternative application manager transfers events between functions in a less timely manner.

C. No Provisioned Concurrency

Provisioned concurrency usually comes at a cost in serverless infrastructures, as function instances have to be kept in memory. For a bursty workload, such as motion tracking triggered by a low-power infrared motion detector, it would thus be tempting not to use provisioned concurrency and rely only on the autoscaling capabilities of serverless edge infrastructures. As our results show, doing so is far from straightforward.

Figure 5 shows results for the previously considered configurations without minimum concurrency. For all configurations we use the simplest autoscaling policy possible, setting a target of 1 instance per in-flight invocation in the resource manager. We use a workload corresponding to approximately 75% of the maximum capacity. In this experiment, we are interested in how long it takes for the system to scale up and converge to stable performance. We run the experiment multiple times for each configuration and show the result of a single representative outcome for each configuration. For the Dedicated variants, we tried both preallocating or not preallocating the single aggregator instance and found that it has no impact on the overall results as the simple aggregator function can quickly catch up once it starts processing detections.

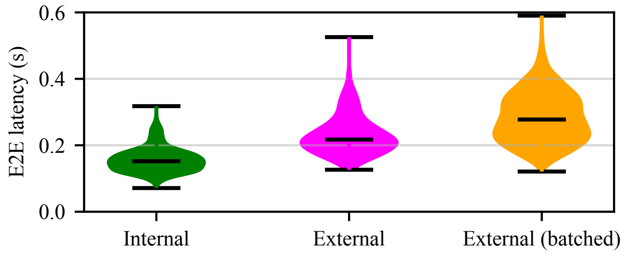

Fig. 6. End-to-end latency for three state management methods using minimal preprocessing and a workload near their respective maximum throughput.

Overloading serverless can cause unstable behavior: As in the previous experiment, we see that reserving vCPUs for the detection instances can impact performance significantly, as seen in the results produced by the `Reserved` configuration. Though the workload is safely below the maximum throughput of this configuration it takes longer to deal with the queue that has been built up while the system is scaling up. In the CPU utilization plot we can see that CPU usage drops down to nearly 0% around the 200-second mark. These drops in CPU usage eventually occur when the resources are overloaded, which causes the resource manager to restart all the instances, severely degrading performance. We are unable to determine exactly why this happens.

More concurrency means more cold starts: Unlike in the case of provisioned concurrency, the `60 Unreserved Dedicated` configuration produced the best results, converging faster than `100 Unreserved Dedicated`. One may expect that the better-performing configuration would stabilize faster but these results reveal the severe costs of cold starts on constrained resources. The lower number of instances means that there are fewer cold starts occurring, and *the cold starts do not just take time but also take up precious resources while the instances are starting*.

In the first experiment we noted that *flow control* occurred for one of the `Shared` variants, which could have played a crucial role in this experiment in preventing overloading the system, but we find no such effect.

D. Minimum Overhead

Figure 6 shows the results for when we replace the workload with tiny 1 pixel images and disable the inference step within the detection function, allowing us to significantly increase the rate of events until the stateful aggregator becomes the bottleneck. We find that the throughput of the aggregator depends greatly on how the state is managed.

Batching greatly increases maximum throughput: We find that both the `Internal` variant (which uses batching) and `External (batched)` can reliably handle 700 events per

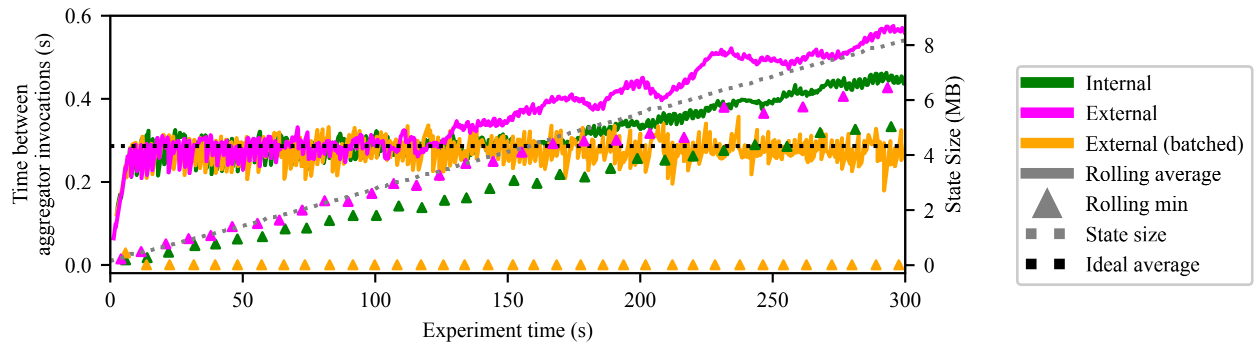

Fig. 7. Average and minimum over a rolling window size of 20 consecutive events handled by the aggregator for three state management methods, using an aggregator with a growing state.

second. In contrast, the `External` variant can only achieve up to 70 events per second, demonstrating how batching significantly alleviates bottlenecks.

E. State Size

Finally, Figure 7 shows the results for when we let the size of the aggregators' state grow linearly over time by retaining all detections received indefinitely. Here we focus on the time in between aggregator function invocations, and use a workload of 0.5 FPS. We send events in groups of 7 events once every two seconds, thus the ideal average latency for time between aggregator events is 2/7 seconds. The perfect system processing such a workload would receive each group of 7 events and process the first one with a latency just below 2 seconds, and then immediately process the following 6 events with minimal latency. Due to variance within the detection function the events arrive to the aggregator out of sync, but in the results for `External (Batched)` we see something close to the ideal pattern, where the rolling minimum remains around a few milliseconds throughout the experiment.

Serialization is expensive: In any system, allowing the state to expand indefinitely will be unsustainable, with the sustainable size being contingent upon factors like the event arrival rate and processing time. However, our findings reveal a critical inefficiency: the rolling minimum for `Internal` increases linearly with the state size. We expect it to behave similarly to the `External (Batched)`, as they all receive events in batches in the same way. With further investigation, we find that the Statefun python function handler serializes and deserializes the state for each event, *even within a batch*, and as the state size grows this serialization/deserialization becomes more and more expensive, such that it eventually becomes impossible to handle events at the same rate as they are arriving in, emphasizing the importance of efficient state management.

V. Discussion

Number of instances and resource sharing: The results in subsection IV-A show that serverless functions can efficiently share compute resources when the resources are not reserved per instance, which is particularly beneficial for multithreaded implementations of ML inference tasks. This is in contrast to the common practice among cloud providers, which reserve resources for function instances, and offer a small set of instance sizes to choose from. Doing so simplifies resource allocation for cloud providers, but leads to resource inefficient deployments [16].

A complementary approach is to allow *instance-level concurrency* to be configured by the user so that a single instance can handle multiple events concurrently. This has been made commercially available in Google Functions and its automation has been investigated in [12].

Batching in stateful and stateless tasks: Previous work has shown that batching can significantly improve the throughput of stateless inference tasks (such as object detection), as inference libraries are typically optimized for batches of inputs [17]. In subsection IV-D and subsection IV-E we found that batching plays a crucial role in the performance of state access as well, as it allows the state to be kept hot in memory for a sequence of events accessing the same state. This is a novel aspect compared to maintaining state locality, i.e., persisting state at the edge location where it is used, considered in previous work [3] [4] [5].

Beyond the importance of batching, our experiments, which consider a single edge location where everything happens locally, show that keeping the state in the instance memory can achieve an order of magnitude faster processing and greater throughput. However, we also saw that maintaining the state purely in memory of an instance, as in the `Unmanaged` variant, quickly failed due to state corruption. The tradeoff between maintaining state in memory and guaranteeing consistency and fault tolerance is a core challenge for stateful serverless, at the edge and in the cloud alike.

Flow control and fine tuning configurations: In subsection IV-A we observed that flow control (i.e. rate limiting) plays a crucial role for one variant to achieve a lower end-to-end latency. The specific configuration of that variant happened to fit with the default configuration of flow control. Within all our experiments we made no effort to fine-tune the configuration for flow control, a non-trivial task which is currently left to the users. Flow control, along with many other configuration parameters should be tuned automatically by the serverless abstraction as hiding the complexity of operations is an essential part of the serverless paradigm [18] [19].

Scale up strategies and cold starts: The only way to avoid cold starts in serverless is to keep instances allocated at all times, but such deployment strategies cancel most of the benefits provided by serverless computing. In cloud environments, the long stabilization period we find in our experiments (subsection IV-C) could be reduced significantly by simply provisioning more resources until the queue stops growing, and then eventually de-provision excess resources [20]. In edge deployments, resources are constrained, and scaling up indefinitely *locally* is not possible, requiring new strategies.

VI. Conclusion

In this study, we examined the feasibility of deploying existing cloud-based serverless software within resource-constrained edge environments. Our results reveal several insights related to local resource management at the edge: to achieve optimal performance maximum utilization of CPU resources becomes imperative. We find that the detrimental effect of cold starts is magnified in limited resource settings, exposing a significant shortcoming of conventional serverless frameworks designed for the virtually limitless computational capacity of cloud environments. This stark mismatch underscores an urgent demand for the development of resource-efficient strategies tailored to serverless at the edge.

Our investigation also highlights the potential of efficient application orchestration as a means to mitigate performance issues and temper bursts through flow control coupled with effective scaling policies. We emphasize that achieving low-latency stateful serverless computing necessitates not only local access to storage but also optimization of state access patterns such as batching. Collectively, these insights demonstrate that all three layers of stateful serverless - application, state, and resource management - require substantial improvements to realize high-performance stateful serverless at the edge.

Acknowledgments

We gratefully acknowledge the financial support for this research provided by the Swedish Foundation for Strategic Research (Resilient Stateful Serverless Computing at the Edge), the Vinnova Center for Trustworthy Edge Computing Systems and Applications (TECoSA), and the European Commission through the Horizon Europe project SovereignEdge.Cognit (grant agreement 101092711).

References

[1] S. Eismann, J. Scheuner, E. Van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "Serverless Applications: Why, When, and How?" *IEEE Software*, vol. 38, no. 1, pp. 32–39, Jan. 2021.

[2] V. Sreekanti, "The design of stateful serverless infastructure," Ph.D. dissertation, EECS Department, Univ. California, Berkeley, 2020.

[3] Z. Xu, L. Zhou, W. Liang, Q. Xia, W. Xu, W. Ren, H. Ren, and P. Zhou, "Stateful serverless application placement in MEC with function and state dependencies," *IEEE Transactions on Computers*, vol. 72, no. 9, pp. 1–14, Mar. 2023.

[4] C. Cicconetti, M. Conti, and A. Passarella, "On realizing stateful faas in serverless edge networks: State propagation," in *Proc. of IEEE International Conference on Smart Computing (SMARTCOMP)*, June 2021, pp. 89–96.

[5] M. Szalay, P. Matray, and L. Toka, "Annabelladb: Key-value store made cloud native," in *Proc. of International Conference on Network and Service Management (CNSM)*, Nov. 2020, pp. 1–5.

[6] H. Zhang, Z. Zhang, L. Zhang, Y. Yang, Q. Kang, and D. Sun, "Object tracking for a smart city using IoT and edge computing," *Sensors*, vol. 19, no. 9, Apr. 2019.

[7] M. de Heus, K. Psarakis, M. Fragkoulis, and A. Katsifodimos, "Distributed transactions on serverless stateful functions," in *Proc. of ACM International Conference on Distributed and Event-Based Systems (DEBS)*, June 2021, p. 31–42.

[8] W. Zhang, V. Fang, A. Panda, and S. Shenker, "Kappa: a programming framework for serverless computing," in *Proc. of ACM Symposium on Cloud Computing (SoCC)*, Oct. 2020, pp. 328–343.

[9] S. Burckhardt, B. Chandramouli, C. Gillum, D. Justo, K. Kallas, C. McMahon, C. S. Meiklejohn, and X. Zhu, "Netherite: Efficient execution of serverless workflows," *Proc. of the VLDB Endowment*, vol. 15, no. 8, p. 1591–1604, Apr. 2022.

[10] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful functions-as-a-service," *Proc. of the VLDB Endowment*, vol. 13, no. 12, p. 2438–2452, July 2020.

[11] P. Raith, S. Nastic, and S. Dustdar, "Serverless edge computing—where we are and what lies ahead," *IEEE Internet Computing*, vol. 27, no. 3, pp. 50–64, May 2023.

[12] S. Li, W. Wang, J. Yang, G. Chen, and D. Lu, "Golgi: Performance-aware, resource-efficient function scheduling for serverless computing," in *Proc. of ACM Symposium on Cloud Computing (SoCC)*, Oct. 2023, p. 32–47.

[13] T. Chavdarova, P. Baqué, S. Bouquet, A. Maksai, C. Jose, T. Bagautdinov, L. Lettry, P. Fua, L. Van Gool, and F. Fleuret, "Wildtrack: A multi-camera hd dataset for dense unscripted pedestrian detection," in *Proc. of IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018, pp. 5030–5039.

[14] J. Ong, B.-T. Vo, B.-N. Vo, D. Y. Kim, and S. Nordholm, "A Bayesian filter for multi-view 3D multi-object tracking with occlusion handling," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 5, pp. 2246–2263, Oct. 2022.

[15] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," Apr. 2018. [Online]. Available: http://arxiv.org/abs/1804.02767

[16] M. Bilal, M. Canini, R. Fonseca, and R. Rodrigues, "With great freedom comes great opportunity: Rethinking resource allocation for serverless functions," in *Proc. of European Conference on Computer Systems (EuroSys)*, May 2023, p. 381–397.

[17] Y. Yang, L. Zhao, Y. Li, H. Zhang, J. Li, M. Zhao, X. Chen, and K. Li, "INFless: a native serverless system for low-latency, high-throughput inference," in *Proc. of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Feb. 2022, pp. 768–781.

[18] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, and D. A. Patterson, "What serverless computing is and should become: The next phase of cloud computing," *Communications of the ACM*, vol. 64, no. 5, p. 76–84, Apr. 2021.

[19] P. Townend, A. P. Marti, I. D. L. Iglesia, N. Matskanis, T. Timoudas, T. Hallmann, A. Lalaguna, K. Swat, F. Renzi, D. Bochenski, M. Mancini, M. Bhuyan, M. Gonzalez-Hierro, S. Dupont, J. Kristiansson, R. S. Montero, E. Elmroth, I. Valdes, P. Massonet, D. Olsson, I. M. Llorente, P. Ostberg, and M. Abdou, "Cognit: Challenges and vision for a serverless and multi-provider cognitive cloud-edge continuum," in *Proc. of IEEE International Conference on Edge Computing and Communications (EDGE)*, July 2023, pp. 12–22.

[20] M. Nazari, S. Goodarzy, E. Keller, E. Rozner, and S. Mishra, "Optimizing and extending serverless platforms: A survey," in *Proc. of International Conference on Software Defined Systems (SDS)*, Dec. 2021, pp. 1–8.