Serverless Computing Lifecycle Model for Edge Cloud Deployments

Kien Nguyen*, Frank Loh*, Tung Nguyen†, Duong Doan‡, Nguyen Huu Thanh†, Tobias Hoßfeld*
*University of Würzburg, Institute of Computer Science, Würzburg, Germany
†Hanoi University of Science and Technology, Hanoi, Vietnam
‡FPT Software LLC, Hanoi, Vietnam
Email:{firstname.lastname}@uni-wuerzburg.de or {thanh.nguyenhuu}@hust.edu.vn

***Abstract*—Serverless computing is a new cloud computing execution model. It offers little operational and administrative overhead, function activation on-demand, and significantly reduced resource consumption. For that reason, it is studied heavily in recent literature by means of handling incoming traffic, applicability for different tasks, and overall performance. Especially the introduction of states, where functionality is kept in CPU or memory can improve performance significantly. However, an in-depth analysis of the resource demand by among others, CPU and memory usage, latency, and energy consumption is still a blank spot in literature. To this end, we propose a serverless lifecycle model with intermediate states and deploy an image processing test setup in the edge cloud to test it. Furthermore, we measure important resource metrics of all states and state transitions of the model. This helps us to answer the question, whether keeping functions available in specific states of a serverless computing instance can improve performance without massive negative influence on the resource requirements.**

***Index Terms*—serverless computing, resource efficiency, energy efficiency, edge computing**

I. Introduction

Every person in the developed world will have at least one interaction with a data center every $18\,\mathrm{s}$ of their lifetime by 2025 [1]. Because of this importance, among others, resource management, scalability, and accessibility of data centers are being steadily improved. One important development was the evolution from bare metal hardware for each task to more virtualization or containerization of applications. This fosters easy decentralization of large data centers or clouds to edge cloud solutions. Meanwhile, such small data center solutions closer to the end user are highly requested. From a service or hardware provider's point of view, it also has many benefits. Applications can be split into microservices and tailored to the end user's or application's demands. These services are dynamically moved as close as possible to the end user, but can also be dynamically deployed, orchestrated, or scaled. This flexibility saves resources and decreases latency. Furthermore, it leads to massive cost reduction since each $100\,\mathrm{ms}$ of latency results in a $1\,\%$ loss in revenue for companies like Amazon [2].

Despite these advantages, microservices are highly complex and require a lot of operation and administration. To this end, serverless computing was introduced [3] as a higher-level framework that eliminates some degree of resource provisioning and management overhead. Additionally, serverless functions are only executed when triggered leading to reduced costs for customers, resource saving for cloud providers, and improved overall performance. However, other potential impacts, such as resource and power leakage when controlling the serverless function lifecycle, especially in resource-constrained environments such as Edge computing are not considered yet. Though, it is crucial to study the lifecycle itself thoroughly [4], [5] to effectively control the impact of serverless functions on resource usage, energy efficiency, or service quality in general.

We close this gap in literature by proposing a serverless-based lifecycle model. With an image processing test setup, we study its performance by means of important resource metrics such as CPU, GPU, and memory usage. Furthermore, we quantify the energy consumption of all states and identify critical states. Last, we investigate among others, the latency for state transitions to identify potential bottlenecks for low-latency applications.

Consequently, our contribution is twofold: first, we propose and implement a state-of-the-art serverless-based lifecycle model with cold and warm intermediate states. Second, we study the resource consumption in different states of this model and for state transitions. We, in particular, study the influence on the resource demand when using these intermediate states, keeping parts of an application available in RAM or CPU. This helps to return to normal operation faster and with less overhead. With this investigation, we can compare resource demand of serverless computing using our model to traditional approaches with only an idle and an active state.

Based on these contributions, we define the following two research questions that are answered in this work.

1) RQ1: What is the resource demand of different cold and warm intermediate states in serverless computing and what are key resources?
2) RQ2: Which state transitions have the largest impact on resource consumption and latency and is there a significant overhead in resource requirements for state transitions with the introduction of intermediate states?

The remainder is structured as follows. Section II summarizes background and related literature. Section III describes the model, the testbed, and the test scenario. Then, Section IV presents the results and Section V concludes.

II. Backgrounds and Related Works

This section summarizes fundamental background information and related literature required to understand this work.

A. Containerization and Microservices

Nowadays, most cloud and edge cloud applications are virtualized and deployed at Virtual Machines (VMs). However, the idea of containerization has been proven to be more agile and resilient than Virtual Machines (VMs), it is becoming the standard for Internet of Things (IoT) or edge-based software deployments [6]. Containerization is a method of Operating System (OS) virtualization that involves packaging an application and its dependencies into a self-contained unit called a "container". These containers share the same kernel as the underlying OS and other containers, allowing more efficient resource utilization and increased portability. To enhance scalability and flexibility in a massive deployment, a container-based software architecture named microservices is widely adopted. It involves decomposing a monolithic application into a collection of small, autonomous services that communicate with each other. Each service is responsible for a specific functionality and can be independently developed, deployed, and scaled. Thus, it allows more flexibility and easier management of complex systems. In the past decade, there has been a significant shift towards microservice-based architectures in the development of traditional and IoT applications [7].

B. Serverless Computing and Edge Intelligence

To overcome management and administration overhead of microservices, serverless computing has emerged in recent years. It promises resource-efficient computing, where the services are packaged as individual functions that are managed and deployed on-demand through short-living containers. While a serverless framework is not necessarily based on containerization, container-based approaches are currently dominating serverless platforms in both industrial (AWS lambda [3] or Azure serverless [8] and open-source frameworks (Knative [9], Fission [10], or IBM OpenWhisk [11]). One reason for this is the advance of container orchestration tools such as Kubernetes [12], Docker [13], as well as value-added services for microservices such as Continuous Integration/Continuous Deployment (CI/CD). From this point forward, the terms serverless function and container will be used interchangeably. Because of its simplicity and performance, serverless computing has already been utilized for many use cases. One example is in the cloud context, where video encoding, API composition, and Map-reduce style analytics can be performed [14]. Furthermore, AI-enabled applications deployed "serverlessly" are studied in many works. Ishakian [15] examined the performance of Deep Neural Network (DNN) inference running over AWS Lambda. While the results demonstrate the feasibility of this approach, service quality by means of latency or throughput may not be guaranteed. Carreira et al. [16] highlight the potential of serverless computing for deploying distributed machine learning (ML) models. Serverless computing provides fine-grained functions that can scale rapidly and removes complexity in resource management and configuration for distributed ML workloads. Similarly, Wang et al. [17] also consider serverless advantages in distributing ML tasks. They test model training with multiple parallel-running serverless functions and prove $4\,\%$ cost reduction on AWS Lambda.

Because of its resource-friendly deployment possibilities, serverless computing has many application areas in an edge cloud environment. In this context, edge intelligence refers to the inhabitant of smart applications in the proximity of data collectors like sensors or cameras. By doing so, it enables low latency for high-demand applications such as automobiles, real-time detection, and health supervision. Edge intelligence can greatly benefit from running on a serverless platform, as the event-driven execution model of serverless computing can enhance resource and power consumption [18]. Rausch [19] showed that deploying DNN models at the edge can benefit from serverless computing in terms of power consumption by placing different layers of the DNN model in different serverless functions, which reside on different devices and only run when triggered. However, due to the high latency intrinsic of serverless, real-time ML still requires further improvement. Patros [20] uses a serverless-based federated learning platform in an edge environment to enable AI power in austere environments, where infrastructure is scarce. However, as serverless is new to the edge, there are many challenges related to Quality of Service (QoS) and resource optimization [5].

C. Serverless' Lifecycle Challenges

One of the key misalignments between serverless and real-time AI applications relates to the lifecycle of serverless functions. In serverless computing, functions are only activated and executed if a specific event occurs. Otherwise, they are either temporarily inactive or shut down. The deployment of new functions, known as "cold start", may require from several hundred milliseconds to many seconds to complete, which can significantly prolong the total response time of a request and violate the service's Service Level Agreements (SLA). To mitigate performance issues, well-known serverless frameworks such as AWS Lambda and Azure Serverless maintain a set number of frequently used containers in a warm state. Various solutions also exist in academia. Mohan et al. [21] propose pre-crafting a warm container for an upcoming task, which can result in an $80\,\%$ reduction in execution time. Mahajan et al. [22] utilize similarities in container content to decrease network requirements for code pulling, thus shortening cold start time. Wu et al. [23] propose a container scheduler that monitors the containers' states (used, paused, or evicted) to reuse them for incoming traffic optimally.

Previously, the lifecycle of serverless functions was often described in a one-way line from service deployment to container termination. The only considered cost was the latency [5]. Wu et al. [23] proposed a lifecycle-aware scheduling scheme named CAS to manage the number of serverless functions that are in idle/paused or null state so that incoming requests can be served with lower latency. Bhattacharjee et al. [24] proposed a state machine that describes the life-

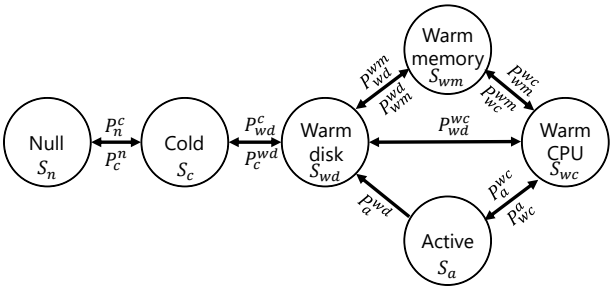


Fig. 1: Serverless lifecycle state machine.

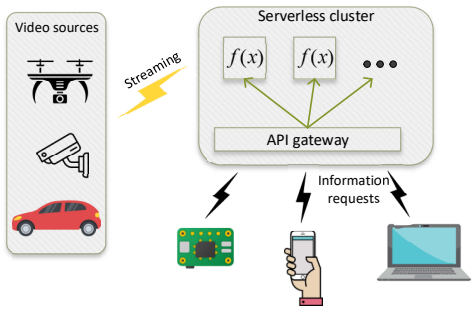


Fig. 2: Image processing use cases of serverless computing.

cycle of a combined service VM and serverless functions leased from the public cloud. However, their model is purely theoretical and only takes the latency cost caused by each state transition into account. Mahmoudi et al. [25] developed analytical models to analyze a serverless lifecycle in terms of average response time, probability of cold start, and the average number of function instances in the steady state. However, similar to other works, they only consider the latency of different states of serverless functions deployed in public cloud environments. In contrast, we consider different resource metrics like CPU, GPU, and RAM usage to evaluate our serverless lifecycle model in this work. In addition, we quantify energy consumption and latency and analyze all metrics through a real testbed in an edge context.

III. Model Proposal and Testbed

In this section, we present a new model for the lifecycle of serverless-based functions and introduce a testbed that is used to measure and analyze different resource metrics.

A. Serverless-based Container's Lifecycle Model

Having a thorough understanding of a serverless function's lifecycle can assist operators and users in evaluating cost and benefits of remaining in a given state or transitioning to another. However, to the best of our knowledge, there is lack of consistency and completeness in the definitions of these states in literature. For that reason, we propose a novel lifecycle as depicted in Figure 1. The presented state machine consists of six states and state transitions between the states. A state is defined as the point, where a function can stay indefinitely. In contrast, a state transition occurs in a fixed period of time to transform the function from one state to another one. The two-directed arrows indicate that both ways are possible to reach. The different states can be described as follows.

Null (S_n) denotes the idle state of the physical machine. Even though serverless platform has been implemented, nothing related to any service exists.

Cold (S_c) denotes that the abstraction/fingerprint of the service has been deployed. The underlying serverless platform has been acknowledged, an API gateway for the service is created, but no image or back-end containers exist yet.

Warm Disk (S_{wd}) is when both, service abstraction and the image for function deployment are available.

Warm CPU (S_{wc}) means one or multiple functions have been deployed. The code inside these containers has been initialized successfully and is ready to receive requests. This state is the most common in traditional container deployment.

Warm Memory (S_{wm}) refers to the pause state of a container. While the code in a container has finished initialization, the Linux cgroup freezer freezes all containers belonging to that cgroup. Thus, all processes are suspended. It is noted that all variables are still stored in RAM.

Active (S_a) state is when the back-end containers received requests and are processing them. Normally, this leads to a surge in resource consumption.

In addition, the state transitions in Figure 1 are symbolized as P_a^b, indicating a process from state a to state b. Each process is fine-grained with no intermediate state occurring within. For example, a transitioning function from Active to Cold requires the execution of two phases: one to traverse back to Warm disk state, and another to remove the image and complete the transition to Cold. For that reason, there is no direct arrow from Active to Cold in the model.

B. Testbed

To study our model, we present a test use case of serverless computing at the edge in the following. We highlight details of the testbed setup and discuss important parameters.

1) Use Case Description: For our test use case, an image processing service is deployed in an edge cloud environment. In a smart city vision, shown in Figure 2, various scenarios such as drones deployed in highly-congested roads or crowded public areas, closed-circuit television (CCTV) for security, or automobiles with embedded cameras are possible. These applications generate data that require smart functions to be processed. In traditional deployments, such smart applications are hosted locally or remotely in an edge or cloud server, and are always ready to process. This is similar to Warm CPU state in our model. When these scenarios are applied in serverless computing, back-end instances can stay at one of the non-Active states, Cold or Warm, as described in Figure 1 until an event is triggered. If such an event, like a request from an end device asking for object or human detection is arriving, they can move to Active state to start processing. Based on this use case, a Concurrent Neural Network (CNN) based object detection is deployed in the form of a serverless

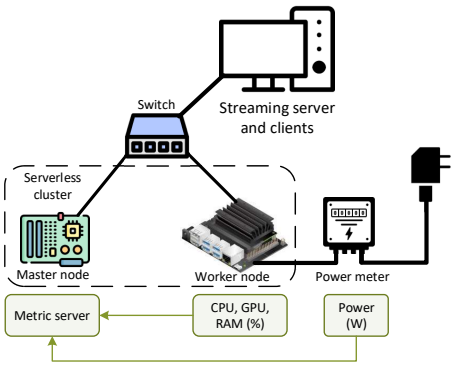


Fig. 3: Testbed setup.

TABLE I: Testbed components overview.

Role	Hardware	Software
Master	Raspberry Pi 4B: 1.5GHz Quad -core Cortex-A72, 4G RAM	Ubuntu 20.04 Kubernetes 1.25 Knative 1.8
Worker	Jetson Nano: 1.43GHz Quad-core Cortex-A57, 128-core NVIDIA Maxwell, 2GB RAM	
Traffic generator	Linux PC	Ubuntu 20.04 Docker 20.10
Measurement	Tinkerforge Voltage/Current Bricklet 2.0	Prometheus 2.34 Curl 7.68

function in our testbed. The CNN model is publicly available at Github [26]. Our serverless function is able to receive HTTP requests, capture a specified streaming video, process each frame by the Neural Network model, and then respond with the number of objects detected in each frame.

2) Testbed Setup: Figure 3 illustrates our testbed for the aforementioned use case. Essentially, it is composed of three main parts, the serverless cluster, the traffic generator, and the measurement tools. The serverless cluster utilizes Knative as its platform, which is based on a master and worker architecture. To emulate a smart edge environment, we employ an NVIDIA Jetson Nano as worker node to execute the serverless functions. It is an embedded device equipped with a GPU. The master device, a Raspberry Pi 4B, manages container scheduling for the Knative cluster and serves as a metrics server for collecting and retrieving log data from resource collectors. To generate streaming sources and HTTP requests from end devices, the testbed utilizes Docker containers hosted on a Linux desktop, referred to as the traffic generator. One container is responsible for streaming a static video of a crowded street, while the other sends HTTP requests to the serverless gateway. All computing devices are connected to an isolated LAN with a maximum bandwidth of $100\,\mathrm{Mbps}$ to avoid bandwidth being the limiting factor. The targeted measurement metrics for the testbed are the consumption of computing resources and includes CPU, GPU, and RAM usage, power consumption, and response time of the serverless functions. The testbed uses Prometheus [27] to monitor resource metrics and an external power meter device for power measurement. To accurately capture a function's state, we directly read the container's event from the Kubernetes API using the Python-Kubernetes library. The same technique is applied to measure the duration of different state transitions. For transitions that are not able to be captured by the library (Warm CPU or Warm Memory to Active), results are obtained using *curl* logging [28]. To synchronize these logging data with each state and state transition of a serverless function's lifecycle, a Python program is executed on the Master node. This program automatically deploys the object detection function, captures function's states and timestamps of each state, and collects logging data from the measurement devices. A summary of the testbed's components is shown in Table I. While state transitions occur for a limited amount of time, states' duration is indefinite. Thereby, we measure state's resources for a period of $120\,\mathrm{s}$. Within this period, every measured metric is collected instantaneously in $0.5\,\mathrm{s}$ intervals. We conducted 50 measurements runs of every state and state transition in the lifecycle. Each run is separated by an idle period of $30\,\mathrm{s}$.

IV. Measurement Results

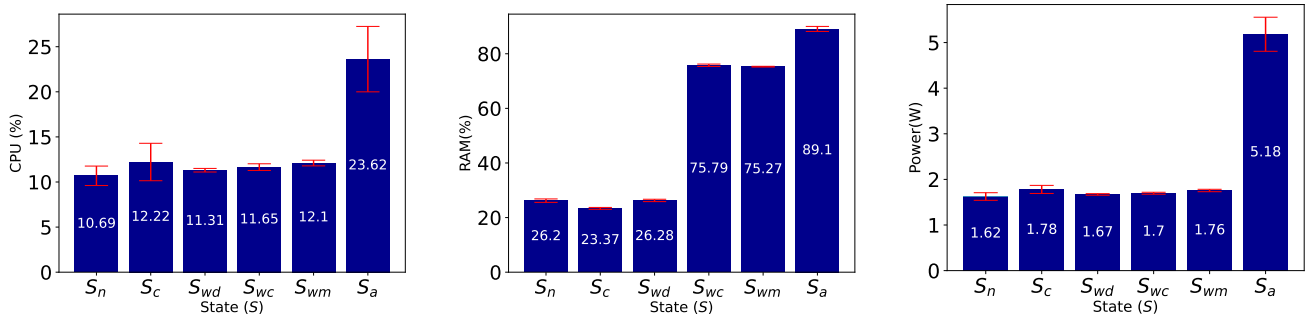
This section presents the results of our testbed measurement for each lifecycle's state and each state transition.

A. State Evaluation

To evaluate the resource demand in each state, Figure 4 shows their CPU, RAM, and power requirements as bar plots with errorbars indicating the $90\,\%$ confidence interval. The y-axis of each subfigure presents the target resource demand metric and each x-axis shows the different states as initialized in Figure 1. Figure 4 shows that even at Null state, the device consumes around $10\,\%$ CPU and more than a quarter RAM capacity due to container and serverless frameworks (Kubernetes, Knative) running in the background. This suggests that a serverless framework with a smaller footprint should be proposed for deployment in the edge. Other states besides

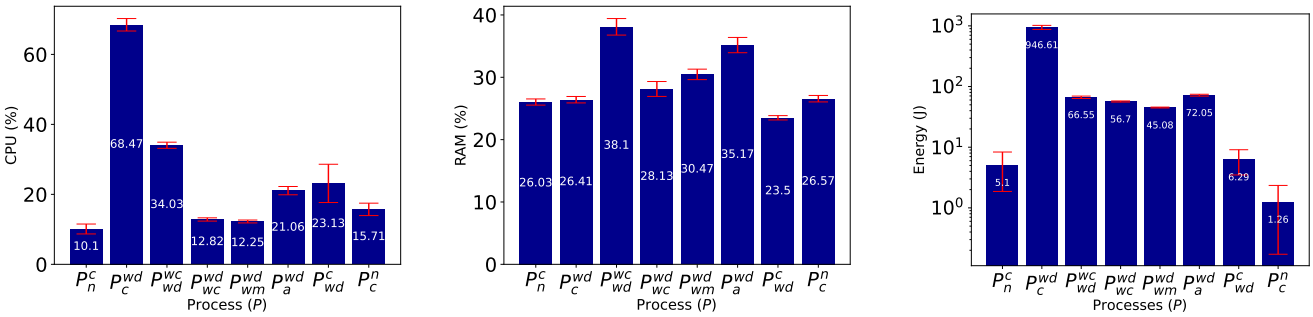
TABLE II: Latency of each state transition.

Transition	Latency (ms)	
	Mean	Std
P_n^c – Null to Cold	3662	1990
P_c^{wd} – Cold to Warm Disk	341705	32764
P_{wd}^{wc} – Warm Disk to Warm CPU	23715	1469
P_{wm}^{wc} – Warm Mem to Warm CPU	110	16
P_{wc}^{a} – Warm CPU to Active	7	1
P_{wc}^{wd} – Warm CPU to Warm Disk	31773	483
P_{wm}^{wd} – Warm Mem to Warm Disk	32761	374
P_{a}^{wd} – Active to Warm disk	34656	2782
P_{wd}^{c} – Warm disk to Cold	3032	1060
P_{c}^{n} – Cold to Null	1193	711



(a) States' CPU usage (b) States' RAM usage (c) States' power usage

Fig. 4: Resources usage in each serverless function's state (errorbars are 90 % confidence intervals).



(a) State transitions' CPU usage (b) State transitions' RAM usage (c) State transitions' energy usage

Fig. 5: Resources usage for each state transitions (errorbars are 90 % confidence intervals).

Active exhibit minimal to no variations in CPU usage since no task is performed. In the Active state, a statistically significant increase in CPU usage to a mean of 23.62 % is detected.

When GPU usage is considered, results are similar to CPU usage results. While the Active state requires 58 %, the requirement for all other states is close to zero as they do not perform any task. In contrast, RAM usage shows distinct trends among them. While Cold and Warm Disk seem to not consume additional memory, Warm CPU and Warm Memory consume almost as much as Active, with a statistically significant increase compared to Cold. There, the detection application has loaded all necessary variables into RAM. RAM utilization increases in a statistically significant way for about 10 % more in Active to load the incoming video.

As power consumption is correlated to computing resource utilization, especially CPU and GPU, it is observed a surge value in Active while the other states remain relatively equal.

The results indicate that non-Active states have a minimal impact on computing resources such as CPU and GPU. In contrast, memory requirements are increased, which is hard-drive in case of Warm Disk and RAM for Warm CPU and Warm Memory. Despite this, these factors have little to no effect on device power consumption. On the other hand, remaining in Warm CPU or Warm Memory state offers a tremendous advantage in latency which will be shown in the following state transition results.

Based on these results, our first research question, RQ1, is answered as follows: *The resource demand in warm states in serverless computing is only influenced slightly. No statistically significant increase is shown for CPU, GPU, and power consumption. The key resource that is increasing is the RAM.*

B. State Transition Evaluation

Since Warm Memory to Warm CPU, Warm CPU to Warm Memory, Warm CPU to Active, and Active to Warm CPU, occur instantly and consume nearly no resources, their resource measurements are omitted in the following. It is also noted that energy consumption is reported instead of power usage as state transitions connect closely with time. The resource consumption of each transition is shown in Figure 5 and the latency in Table II. Considering resource allocation states (P_n^c, P_c^{wd}, P_{wd}^{wc}, P_{wm}^{wc}, P_{wc}^a), the state transition from Null to Cold (P_n^c) consumes the least resources and energy while taking almost 4 s to finish. Thus, it is recommended that every service should immediately move to Cold state when they're deployed regardless of traffic to prevent additional delay. Furthermore, no statistically significant additional resources are required in Cold state as seen above. In contrast, Cold to Warm Disk (P_c^{wd}) has the highest resource consumption, with high CPU requirements for image download and extraction. These tasks have also contributed to a long latency of more than 300 s, making it the longest and most energy-intensive process.

The container initiation process (P_{wd}^{wc}) also consumes a large amount of CPU and RAM, and a considerable amount of time with $23\,\mathrm{s}$. During this state transition, the application inside the container must be started and resources are allocated. When comparing these results to the state's results, staying at Warm Disk can save $50\,\%$ RAM but will suffer from large cost if a request arrives later and require the function to move to Active. In our use case, this cost sums up to $6\,\mathrm{min}$ delay and $1000\,\mathrm{J}$ energy. Therefore, if memory is not a concern, it is preferable to keep serverless functions in Warm CPU or Warm Memory state, especially when these states have a small impact on computing resources and energy consumption.

Comparing Warm CPU to Warm Memory, activating the application from Warm CPU only takes about $7\,\mathrm{ms}$ and $110\,\mathrm{ms}$ for Warm Memory. Our tests show no statistically significant resource consumption improvement of Warm Memory over Warm CPU. Thus, Warm CPU should be preferred as it provides a ten-fold latency reduction, which is crucial for real-time applications. The resource-deallocation processes (P_{wc}^{wd}, P_{wm}^{wd}, P_{a}^{wd}, P_{wd}^{c}) consume less resources and have no effect on the application's quality. However, in large scenarios where hundreds of functions are deployed on a single physical machine, the constant transitions between states may lead to much higher energy consumption and bottlenecks of specific resources for other active functions. Based on these findings, we can answer our second research question as follows: *Out of all state transitions, Cold to Warm Disk and Warm Disk to Warm CPU have the largest impact on resource consumption and latency. As a result, there is a significant overhead in these state transitions, especially the Cold to Warm Disk.*

V. Conclusion

Serverless computing can simplify operation, management, and administration of functions in cloud and edge cloud computing. Especially the introduction of warm states, where specific functionality is kept in memory or CPU, can improve reaction time and latency for services significantly. However, until now, the influence on resource consumption like memory or CPU usage and energy consumption is not fully analyzed. We close this gap in literature by proposing a serverless lifecycle state machine, set up a testbed, and measure an edge cloud use case. We measure the resource and energy consumption of all relevant states and state transitions and study the latency. While we see a statistically significant increase in RAM usage in warm states, CPU, GPU, and power requirements are not influenced. For that reason, we conclude that if memory resources allow it, keeping a warm state is always beneficial. The same is true when analyzing state transitions, where a significant latency reduction is possible. With our results, researchers and operators can define a strategy for their serverless function's lifecycle to improve a specific objective or multiple objectives. Since our testbed has been conducted on a one-function scenario, a further examination into multi-function deployment can be considered in future works to generalize the influence of each state and process on resource consumption and application performance.

References

[1] Danfoss. Data Center Sustainability - a More Efficient Future is Needed. [Online]. Available: https://www.danfoss.com/en/about-danfoss/insights-for-tomorrow/integrated-energy-systems/data-center-power-consumption/

[2] FastCompany. How One Second Could Cost Amazon $1.6 Billion In Sales. [Online]. Available: https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales

[3] Amazon. Serverless Computing - AWS Lambda - Amazon Web Services. [Online]. Available: https://aws.amazon.com/lambda/

[4] P. Patros, J. Spillner, A. V. Papadopoulos, B. Varghese, O. Rana, and S. Dustdar, "Toward Sustainable Serverless Computing," *Internet Computing*, 2021.

[5] R. Xie, Q. Tang, S. Qiao, H. Zhu, F. R. Yu, and T. Huang, "When Serverless Computing Meets Edge Computing: Architecture, Challenges, and Open Issues," *Wireless Communications*, 2021.

[6] R. Cziva and D. P. Pezaros, "Container Network Functions: Bringing NFV to the Network Edge," *IEEE Communications Magazine*, 2017.

[7] I. M. Development and Insights, "Microservices in the Enterprise, 2021: Real Benefits, Worth the Challenges," 2021.

[8] Azure Serverless. [Online]. Available: https://azure.microsoft.com/en-us/solutions/serverless/

[9] Knative. Serverless Containers in Kubernetes Environments. [Online]. Available: https://knative.dev/docs/

[10] Fission. Fission: Serverless Functions for Kubernetes. [Online]. Available: https://github.com/fission/fission

[11] Apache. Open Source Serverless Cloud Platform. [Online]. Available: https://openwhisk.apache.org/

[12] Kubernetes. Kubernetes: Production-Grade Container Orchestration. [Online]. Available: https://kubernetes.io

[13] I. Docker. Docker. [Online]. Available: https://www.docker.com

[14] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The Rise of Serverless Computing," *Communications of the ACM*, 2019.

[15] V. Ishakian, V. Muthusamy, and A. Slominski, "Serving Deep Learning Models in a Serverless Platform," in *Int. Conf. on Cloud Eng.*, 2018.

[16] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "A Case for Serverless Machine Learning," in *Workshop on Systems for ML and Open Source Software at NeurIPS*, 2018.

[17] H. Wang, D. Niu, and B. Li, "Distributed Machine Learning with a Serverless Architecture," in *Conf. on Computer Comm.* IEEE, 2019.

[18] S. R. Chaudhry, A. Palade, A. Kazmi, and S. Clarke, "Improved QoS at the Edge using Serverless Computing to Deploy Virtual Network Functions," *Internet of Things Journal*, 2020.

[19] T. Rausch, W. Hummer, V. Muthusamy, A. Rashed, and S. Dustdar, "Towards a Serverless Platform for Edge {AI}," in *USENIX Workshop on Hot Topics in Edge Computing*, 2019.

[20] P. Patros, M. Ooi, V. Huang, M. Mayo, C. Anderson, S. Burroughs, M. Baughman, O. Almurshed, O. Rana, R. Chard *et al.*, "Rural AI: Serverless-Powered Federated Learning for Remote Applications," *Internet Computing*, 2022.

[21] A. Mohan, H. Sane, K. A. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile Cold Starts for Scalable Serverless," in *HotCloud*, 2019.

[22] K. Mahajan, S. Mahajan, V. Misra, and D. Rubenstein, "Exploiting Content Similarity to Address Cold Start in Container Deployments," in *Int. Conf. on Em. Networking Experiments and Technologies*, 2019.

[23] S. Wu, Z. Tao, H. Fan, Z. Huang, X. Zhang, H. Jin, C. Yu, and C. Cao, "Container Lifecycle-Aware Scheduling for Serverless Computing," *Software: Practice and Experience*, 2021.

[24] A. Bhattacharjee, A. D. Chhokra, Z. Kang, H. Sun, A. Gokhale, and G. Karsai, "Barista: Efficient and Scalable Serverless Serving System for Deep Learning Prediction Services," in *International Conference on Cloud Engineering*. IEEE, 2019.

[25] N. Mahmoudi and H. Khazaei, "Performance Modeling of Serverless Computing Platforms," *Transactions on Cloud Computing*, 2020.

[26] Nvidia. Deploying Deep Learning. [Online]. Available: https://github.com/dusty-nv/jetson-inference

[27] Prometheus. Prometheus - Monitoring System & Time Series Database. [Online]. Available: https://prometheus.io/

[28] Curl. Curl - How to Use. [Online]. Available: https://curl.se/docs/manpage.html