2019 IEEE 12th International Conference on Cloud Computing (CLOUD)

Towards Latency Sensitive Cloud Native Applications: A Performance Study on AWS

István Pelle[1], János Czentye[1,2], János Dóka[2], Balázs Sonkoly[1,2]
[1]MTA-BME Network Softwarization Research Group, [2]Budapest University of Technology and Economics
e-mails: {pelle,czentye,janos.doka,sonkoly}@tmit.bme.hu

***Abstract*—Microservices, serverless architectures, cloud native programming are novel paradigms and techniques which could significantly reduce the burden on both developers and operators of future services. Several types of applications fit in well with the new concepts easing the life of different stakeholders while enabling cloud-grade service deployments. However, latency sensitive applications with strict delay constraints between different components pose additional challenges on the platforms. In order to gain benefit from recent cloud technologies for latency sensitive applications as well, a comprehensive performance analysis of available platforms and relevant components is a crucial first step. In this paper, we address one of the most widely used and versatile cloud platforms, namely Amazon Web Services (AWS), and reveal the delay characteristics of key components and services which impact the overall performance of latency sensitive applications. Our contribution is threefold. First, we define a detailed measurement methodology for CaaS/FaaS (Container/Function as a Service) platforms, specifically for AWS. Second, we provide a comprehensive analysis of AWS components focusing on delay characteristics. Third, we attempt to adjust a drone control application to the platform and investigate the performance on today's system.**

I. Introduction

Microservices, serverless architectures, Function as a Service (FaaS) and cloud native programming are included in the list of "hottest topics" of cloud computing. These paradigms and related techniques, without a doubt, could significantly reduce the burden on both developers and operators of future services. Several types of applications fit in well with the new concepts and techniques easing the life of different stakeholders while enabling novel, cloud-grade service deployments. Resource and life-cycle management, on-demand resource scaling, controlled resiliency and dependability, fault tolerant operation are just highlighted tasks which could easily be passed to the underlying cloud provider. However, latency sensitive applications pose additional challenges on the platform providers. When strict delay constraints are given between different components of, e.g., a microservice based software product, or between a software element and the end device, further considerations are required. Cloud edge, fog, and mobile edge computing are novel concepts extending traditional cloud computing by deploying compute resources closer to customers and end devices. Is it enough to meet latency requirements? The answer is not trivial as the cloud platform itself could significantly contribute to the end-to-end delay depending on the internal operations, involved techniques and available configurations.

In order to gain benefit from recent cloud technologies for latency sensitive applications as well, a comprehensive performance analysis of available platforms and relevant components is crucial. In this paper, we do the first steps towards that analysis to get a better understanding of the capabilities and characteristics of current solutions. We address one of the most widely used and versatile cloud platforms, namely, Amazon's AWS (Amazon Web Services). Besides several features, it supports CaaS (Container as a Service), FaaS (Function as a Service) solutions and several data store options. In case of microservice based distributed applications, several components could have impact on the end-to-end processing delay. On the one hand, runtime of the functions, runtime environments implemented as containers, the amount of data to be transferred between the components and the service structure itself are determined by the software developer. On the other hand, runtime environments provided by FaaS solutions, function invocation time, read and write latency of different data stores are affected by the cloud platform and requested services. For example, AWS provides a diverse set of data store options with different performance characteristics and pricing schemes.

Our contribution is threefold. First, we define a detailed measurement methodology for CaaS/FaaS platforms, more specifically for AWS. Second, we provide a comprehensive analysis of AWS components focusing on delay characteristics which could affect the performance characteristics of latency sensitive applications and users' QoE (Quality of Experience). Third, we attempt to adjust a drone control application to the AWS platform and investigate the performance on today's system.

The rest of the paper is organized as follows. In Sec. II, an illustrative service example and a brief description on the AWS platform are provided. In Sec. III, we introduce the key metrics to be taken into account when building latency sensitive cloud applications, and we present our methodology on gathering such metrics using AWS. Section IV is devoted to the main results gained from the performance measurements. The values we measured characterize the current state of AWS but these might change quickly as new features and better equipment are added to the system. Based on our experiences, we tweak a drone control application and analyze its performance on AWS in Section V. A short summary on the related work is given in Section VI, while Sec.VII concludes the paper.

2159-6190/19/$31.00 ©2019 IEEE
DOI 10.1109/CLOUD.2019.00054

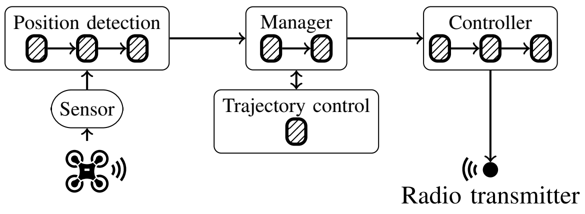


Figure 1: A possible decomposition of drone control.

II. Background

A. Drone Control: A Latency Sensitive Application

Controlling a drone is a typical application where quick reaction time is necessary. This holds true for cases when flying the drone by hand as well as using an automated system for the control. The latter method is preferable when handling multiple drones at the same time e.g., in an industrial setting. Fig. 1 shows a possible layout for performing such a task programmatically. First, the position of the drone needs to be determined in the physical space. Here diverse methods can be used, e.g. real-time image processing or sensory data from GPS or indoor positioning solutions. This information then needs to be combined with a desired path configured using a *Trajectory control* interface. With current and desired position paired by a *Manager* entity, a *Controller* determines the signals to be sent to the actuators. It is evident that the control loop is delay bounded. This time limit is mainly affected by the capabilities of the drone and environmental conditions as well. Having a fast drone or a quickly changing environment requires quick control decisions and low end-to-end latency. Clearly, if control equipment is kept close to the actual moving hardware, these requirements are easily satisfied. Having an on-premise cloud could guarantee fast response times as well as scalability but would require substantial investment for the equipment. Public cloud offerings can alleviate this pain, since users can rent their compute capacity on many different levels of granularity. Choosing such options can enable high scalability which is beneficial when controlling a drone fleet using the same code base or when devices connect to the control loop in an on-demand fashion. In both cases, when new connections are established, the cloud platform can start up new instances to satisfy the increased load. In case of a disconnection, resources can be automatically freed up to reduce price. Besides the clear benefits, however, would such services be able to handle the strict latency requirements as well?

B. Amazon's AWS Platform

AWS's CaaS and FaaS solutions differ in ease of use and price. The first one enables users to run application components in highly customizable containers but thus makes configuration harder. FaaS tries to solve the configuration issue by providing fixed execution environments where small, stateless functions could be run. Before moving on to highlight the main features of the above two services, we note that for most AWS services, users can specify hardware or virtual resources in terms of vCPU (virtual processor), memory size, network performance, size of stored or transferred data. Either one or more of these resources can be configured for a service either one-by-one or using predefined bundles. For the sake of simplicity, in the rest of the paper, we refer to concrete sets of them as (resource) flavors.

AWS Elastic Container Service (ECS) lets users run their own Docker containers on AWS Elastic Compute Cloud (EC2) instances. Here, multiple layers group together containers and resources which are defined in terms of vCPU and memory size as well as EC2 instances. Options for scaling and load balancing are provided as well. Two scaling policies can act based on different metrics. Target tracking keeps a chosen metric around a target value while step scaling sets bounds that initiate scaling. Depending on the chosen configuration, setting up ECS can turn out to be a tiresome effort.

Lambda, i.e. the FaaS offering, executes user code written in various languages (including C#, Java, Node.js and Python) with predefined runtimes. The service takes care of configuring the runtime environment, networking and scaling automatically. Flavors can be selected based on memory size alone which can be adjusted in small increments and the platform assigns vCPUs based on this, in an undisclosed manner. A multitude of event sources is available that can trigger a Lambda function. Triggering a function spins up a new instance whenever there is none to process the event immediately. Instances are stopped automatically after a platform determined amount of time.

Our drone control application can be mapped to AWS resources in several ways. A possible option could be to map bigger components (big boxes in Fig. 1) to ECS instances. Another possibility is to break up these components to small parts (striped boxes in the figure) and run them as Lambda functions. This approach could require state externalization which can be implemented by different data store services of the platform (e.g., Redis, Memcached, DynamoDB, Kinesis). Any combination between these two extremes can be realized as well. In our latency sensitive use-case the main question is: would we be able to keep end-to-end latency requirements? To answer this, a deep understanding is needed about the repercussions of design choices: selecting decomposition methods, services and flavors.

III. Measurement Methodology

When creating a latency sensitive application, the key performance indicator is time which constitutes of two factors: the execution time of each component and the invocation delay between them. Execution time is determined by software (execution environment) and hardware (CPU performance and memory size) capabilities of the underlying platform. In case of serverless applications, the read and write performance of state stores can also have significant impact on the overall execution time. Invocation delay depends on network and platform specific parameters. Network delay is mainly determined by the physical distance between the end points, the internal networks, as well as the capabilities of the physical and virtual traffic forwarding devices. In case of a public cloud provider, end users cannot adjust these factors, however, the method to be used for data transfer and invocation can be chosen.

AWS provides different tools that support information gathering. CloudWatch Logs is available in every region and can collect and display log messages from AWS services and user code. CloudWatch Logs combined with user implemented timestamping and the SDK's synchronous call can be used for fetching measurement data from Lambda functions. X-Ray can sample executions and measure time differences between user defined code segments. We tested both methods of proprietary timestamping and X-Ray, and found that they give identical results. We combined these three tools in different ways to carry out performance tests. Here, we summarize our measurement scenarios and the methodology, while results are given in Section IV.

A. Hardware Configurations

In case of Lambda, five AWS regions are tested. Four of them are in the EU: *eu-central-1* in the Frankfurt region, *eu-west-1* in Ireland, *eu-west-2* in London and *eu-west-3* in Paris. An additional one, located in N. Virginia, USA (*us-east-1*) is tested as well. In case of ECS, only the *eu-west-1* region is tested. Information is gained in each case by reading `/proc/cpuinfo` and logging its contents with CloudWatch Logs.

B. Execution Time Tests

Lambda and ECS performance is benchmarked by executing the same single-threaded Python 3.6 function over different flavors. Our benchmark function is a computation intensive function as it calculates the 28th Fibonacci number while consumes only 54 MB of memory in case of Lambda. ECS containers with two different base images are compared: Python 3.6 and Amazon Linux. They were selected based on their availability on Docker Hub and ability to run code written using Python 3.6. Additionally, Amazon Linux is the operating system that powers the VMs (Virtual Machine) running Lambda functions as well. We note here that, at the time of the test, Lambda uses version 2017. 03. of Amazon Linux, while in our ECS tests, the latest available version of 2018. 03. is used. However, this difference in versions has much less significance on the performance compared to the effect of ECS and Lambda using different virtualization environments, Docker and AWS's own Firecracker platform, respectively. For collecting execution times, X-Ray is used. In case of Lambda, we start with the smallest flavor (128 MB) and double the memory size in each step except when reaching the maximum amount of memory (3008 MB). Execution times are measured for the different CPU types found by our tests described previously. In case of ECS, the memory is adjusted so that it matches the Lambda memory size and the highest vCPU count is selected.

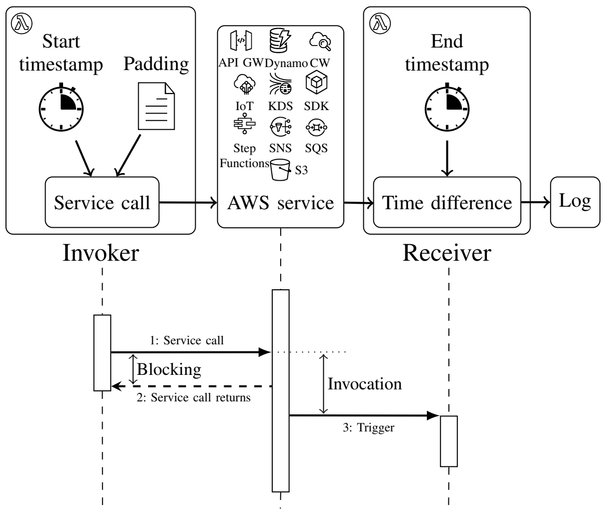


Figure 2: Setup and timing relationships for testing invocation delay. Blocking of the Invoker function is depicted in the case of using an asynchronous call.

C. Invocation Tests

Invocation tests are performed using a setup shown in the top part of Fig. 2 for the event sources listed in Table I. An *Invoker* Lambda function is used to generate a timestamp together with varying-sized padding and send them to a *Receiver* function via a call to different AWS services. When the Receiver function is triggered, it measures the time elapsed since the start of the AWS service call (see the bottom part of Fig. 2) and logs it. Such measurements are valid since AWS Lambda is configured to synchronize to stratum 3 clocks. Thus, in our measurements, invocation time constitutes of data upload to an AWS service, the delay caused by the service triggering the Receiver function and the delay added by the Lambda framework to call the handler function. This methodology has been chosen since most AWS services do not provide accurate enough time stamping solutions for measuring the effective invocation time: the time that passes between the point where the AWS service has all the data the Invoker function sent and the point when the Receiver function starts execution. For the Invoker function, the highest flavor

Table I: Tested data storage (DS) and event source (ES) options for AWS Lambda. For more details, see [1], [2].

Service	Usage	Maximum Data Size
API Gateway	ES	6.2 MB
DynamoDB	ES & DS	400 kB
CloudWatch Logs	ES	262 kB
ElastiCache	DS	Memcached: 1 MB; Redis: 6.8 MB
IoT Core	ES	Async publish: 130 kB
Kinesis Data Streams (KDS)	ES & DS	1048 kB
SDK	ES	Event type: 130 kB; RequestResponse: 6.2 MB
Step Functions	ES	32 kB
Simple Notification Service (SNS)	ES	262 kB
Simple Queue Service (SQS)	ES	Standard queue: 262 kB
Simple Storage Service (S3)	ES & DS	5 GB

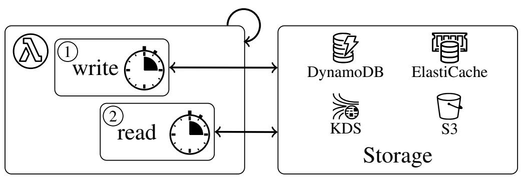


Figure 3: Setup for testing data store access.

(to have the shortest data upload delay) while for the Receiver, the smallest is used (since it has no effect on the total delay). Since both functions are set up in the same AWS region, the configuration minimizes the impact of network delay. For each payload size, 100 tests are run and the mean and standard deviation of the invocation delay are calculated using the same warmed up instances of the Lambda functions. Invocations are performed so that no throttling or batching of more than one element can happen. To compare function invocation delay to a reference, we created a scenario where timestamp and padding data are transmitted between two containers in different VPCs (Virtual Private Cloud) using a STREAM socket. We argue that this use-case is a good reference since traffic has to pass a NAT gateway thus emulating network conditions between Lambda functions. Therefore, additional delay stems from the used event source and the Lambda framework itself. It is worth noting that socket communication does not impose any limit on the size of data to be transmitted while AWS services have such limitations (see Table I).

D. Data Store Access Tests

The co-operation of stateless Lambda functions requires state externalization in several cases. Therefore, the performance of data store related operations can be crucial. AWS offers different types of data stores with different features and performance characteristics. The maximum amount of data which can be stored by a given service is shown in Table I. Performance-wise, in case of S3, there is no parameter that can be set in order to increase speed. In case of KDS, multiple shards can be used to store data but values set for the same key are always stored in the same shard. Reading the same shard is only possible 5 times in a second. In case of DynamoDB, read and write capacity units can be set to determine where AWS throttles incoming or outgoing requests. For ElastiCache services, a flavor can be selected to control the allocated resources (vCPU count, memory size and network performance) of the instances running either Redis or Memcached nodes. We examine the first two options shown in Table II: *t2.micro*, the cheapest one and *r5.large*, a memory optimized configuration. Both services are run using a single node only. 100 measurements are carried out using the setup shown in Fig. 3. X-Ray is used for timing the measurements that first write random string data to the same key (step ①) and then immediately read the data back (step ②).

E. Data Store Throughput Tests

ElastiCache services, based on in-memory data store technologies, are important candidates when we target latency critical applications. As we have seen, the performance can be controlled by selected flavors which determine computation and network performance of the nodes running either Redis or Memcached. This configuration has direct impact on the number of concurrent functions that can be run to access the same storage. We set up tests in order to investigate how throughput and latency characteristics depend on the number of parallel functions accessing the storage. A special Lambda function is constructed that is able to either write or read a key in the specific in-memory storage. Then multiple calls are initiated to instruct AWS to run multiple instances of the same function concurrently. The operation is illustrated by Fig. 4. Each function repeats the same operation until a timeout of 3 minutes is reached. In case of the write operation, 1 kB of random string data is written in the store. With both operations, the latency they impose is measured and the number of successful executions is counted. These metrics are then averaged over all the concurrently running functions. At the end, the total throughput is calculated by multiplying the number of executions by the size of the data. Aggregated metrics are recorded with CloudWatch Logs. Tests are run both with Redis and Memcached using the flavors shown in Table II: *cache.t2.micro* is the smallest flavor to be chosen, while *cache.r5.large* provides good network performance and *cache.m4.2xlarge* offers high processing capabilities.

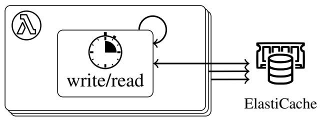


Figure 4: Measurement setup for ElastiCache.

IV. AWS Performance Measurements

In this section, we present our main results on the performance analysis following the previously detailed methodology. Most of the plots depicting delay characteristics show statistical average and standard deviation calculated on data sets excluding outliers[1]. All our tests were performed between November 2018 and January 2019.

A. Hardware Configurations

By conducting the measurements discussed in Section III-A, we found that different regions apply different types of Intel Xeon CPUs to execute Lambda functions. Results show that regions, except for eu-west-3, change processor types at least once when increasing the memory size assigned to a Lambda

[1]In some experiments, we found a few extreme outliers, which cannot be avoided in a public cloud environment.

Table II: Tested ElastiCache node types.

Cache Node Type	vCPU	Memory (GiB)	Network Performance
cache.t2.micro	1	0.555	Low to Moderate
cache.r5.large	2	13.07	Up to 10 Gigabit
cache.m4.2xlarge	8	29.70	High

function. vCPUs in these regions are assigned so that below 512 MB of memory, 1 core – 1 thread is exposed while above that threshold, 1 core – 2 threads can be detected. In case of eu-west-3 there is always 1 core and 2 threads present. A strange observation is that once a CPU type change happens as a result of increasing memory size, Lambda seems to stick to that type. This behavior was observed when decreasing the amount of memory as well as starting new functions in the same region (with low memory sizes). We attribute this behavior to changes in the underlying VM. Default CPU assignment is reset after about a day of not requesting a function with more than 512 MB of memory.

B. Execution Time Tests

Fig. 5 shows run time of a test function executed in container and as Lambda, respectively, following the methodology detailed in Section III-B. In case of Lambda, the resource flavor is defined by the memory size, whereas for a container, the requested vCPU can also be configured (shown by the labels in the plot). We can see that up until 1024 MB of memory, Lambda is able to approximately half execution time when doubling memory size. We found that the shortest execution time is reached at 1792 MB memory where it is reduced by 32% and above that it stays around the same value. As we use single threaded code and the platform controls the allocated fractions of CPU via cgroups (or similar methods), the results are not surprising. We can conclude that a full CPU core is allocated at this memory value. Comparison of the two container base images shows that Amazon Linux executes code faster than Python 3.6 thus in a latency critical case, adequate selection of container image is key. Peak performance in case of both containers is reached at 4 GB of memory with 2 vCPUs. It is more interesting to compare the two services, i.e., Lambda and ECS. For example, the average execution time of a Lambda function with 1024 MB memory is around 4% better than an Amazon Linux container with 2048 MB memory and 1 vCPU. In our test environment,

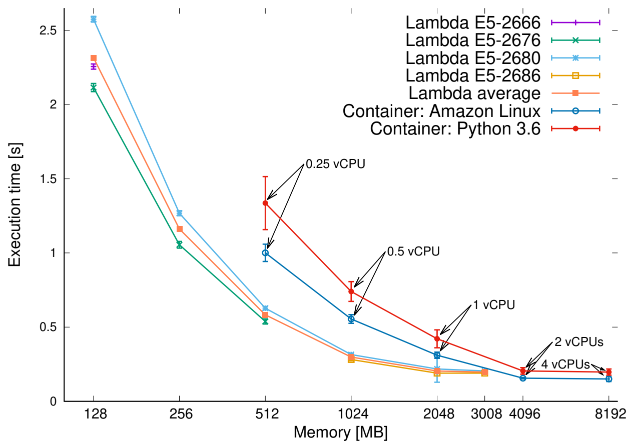


Figure 5: Execution time of the same function using different flavors of AWS Lambda and ECS as runtime environments.

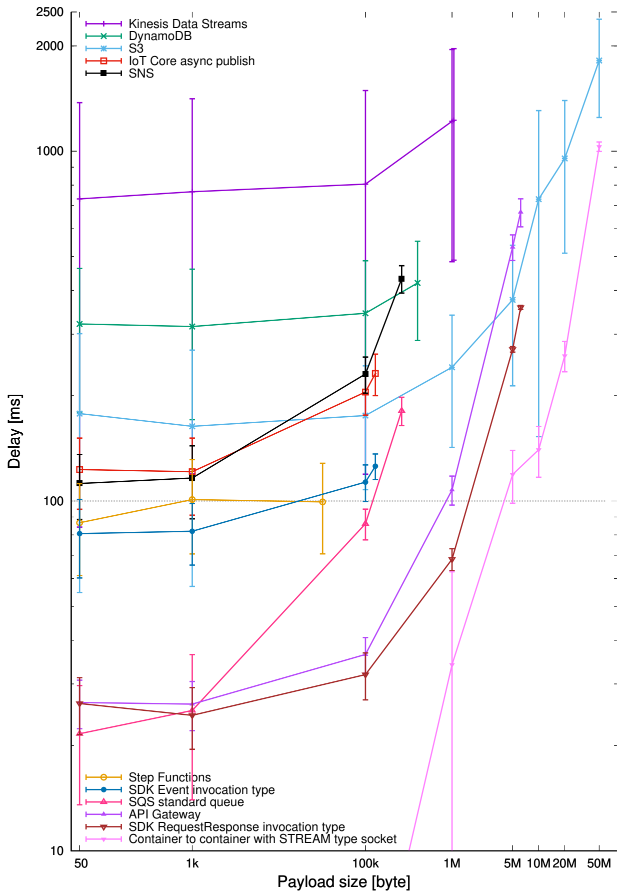


Figure 6: Invocation delay between Lambda and container. For simplicity, the delay of socket communication is omitted under 1 MB of payload size. It is, however, 2.3 ms at 100 kB, and less than 0.4 ms at smaller payload sizes with small variance.

we found that one physical core is inevitably allocated for a container if we configure 2 vCPUs.

C. Invocation Tests

Results of tests discussed in Section III-C are shown in Fig. 6. We can observe that different event sources impose highly different invocation delays and high variances. Our reference measurement of communicating between two containers via a stream socket always performs significantly better than the best Lambda event source options. For payloads with size of maximum 1 kB, SQS and API Gateway event sources as well as synchronous SDK calls (RequestResponse) have the lowest invocation delay. Compared to the reference measurement's less than 0.4 ms delay (not shown in the figure for simplicity) they impose approximately 25 ms. This should come as no surprise since calls use HTTP over TLS. We also note here that API gateway and SDK RequestResponse calls block the Invoker function for a time until the Receiver function returns (which can be significantly longer than the one shown in Fig. 2). This might have an adverse affect

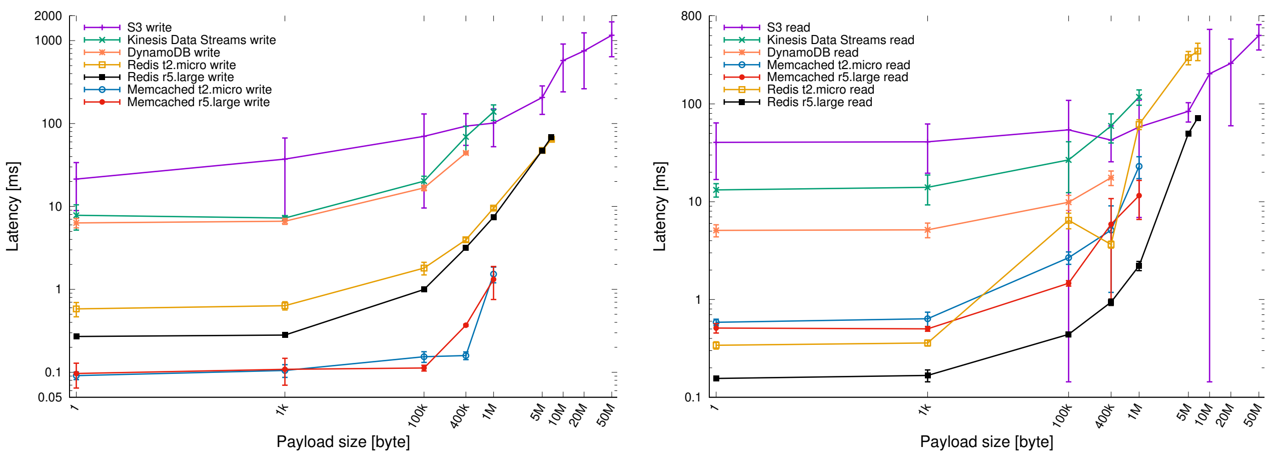


Figure 7: Latency of write (left) and read (right) operations using different AWS data storage services.

on resource usage of the complete application. The SDK's Event type invocation and Step Functions' invocation delay are in the same range, slightly below the 100 ms mark. Although IoT Core provides a convenient way for sending data to Lambda functions from a local IoT device (in this test case running as a container in the same AWS region as the Receiver function), it triggers functions quite slowly. The used MQTT (Message Queuing Telemetry Transport) protocol, message broker and AWS's rules engine impose 100–200 ms of invocation delay that puts the service on the same level as SNS, although the first is able to carry smaller packets. As opposed to other services, S3 passes only an identifier to the Receiver function, not the complete invocation payload. Thus a supplementary read from the given S3 bucket has to be performed for accessing the data which imposes additional delay that is shown on the right side of Fig. 7. KDS, which is designed for real time operations, can trigger a function in 700–1000 ms. This is most likely due to the poll-based trigger mechanism where each KDS shard can be read only 5 times in a second. From 100 kB payload size, the data transmission dominates the overall performance. Based on these measurements, we can conclude that latency sensitive applications are hard to be built on the Lambda platform, since chaining functions quickly adds high delays to the application as well as significant jitter.

D. Data Store Access Tests

Data in Fig. 7 was gained by running the test described in Section III-D. For the write operation (see left side of figure), as expected, S3 is the slowest option. KDS and DynamoDB are slightly better and have similar characteristics. Compared to them, Redis can achieve a tenfold increase in speed for data less than 100 kB in size, while Memcached is even quicker. Comparing different node flavors shows that, for smaller data sizes, Redis is able to achieve a significant speed increase on the bigger flavor while Memcached write speeds do not increase. In general, delay variance can be reduced by bigger flavors.

The read latency results, presented in the right side of Fig. 7, exhibit similar characteristics. Redis is now the quickest with 0.1–0.6 ms of latency for small data sizes. Memcached also shows comparable performance, however, it is sensitive for the delay between write and read operations. More exactly, if we read the data store within 30 ms after the write operation, Memcached behaves erratically as average latency quickly jumps to around 42 ms, and two operation modes can be identified. We believe this behavior is due to the management of multiple threads Memcached handles and the read operation competes with the write process. The presented results for Memcached in the figure correspond to scenarios where the waiting time between write and read operations is 30 ms. Fig. 8 presents the cumulative distribution functions (cdf) for selected scenarios of Redis and Memcached. We observed high variance in case of Memcached at certain data sizes above 50 kB. They were caused by the fact that the majority (60–80%) of the measurements were centered around a value but the rest were spread out over a much greater time period sometimes spanning over tens of milliseconds. We believe this behavior is again due to thread management. S3 shows similar characteristics at each measurement point.

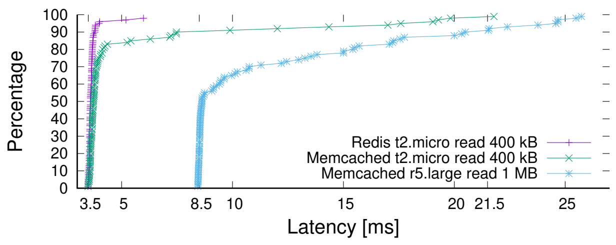
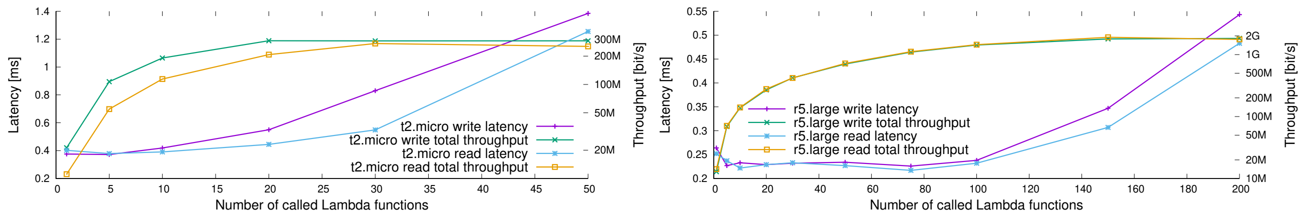
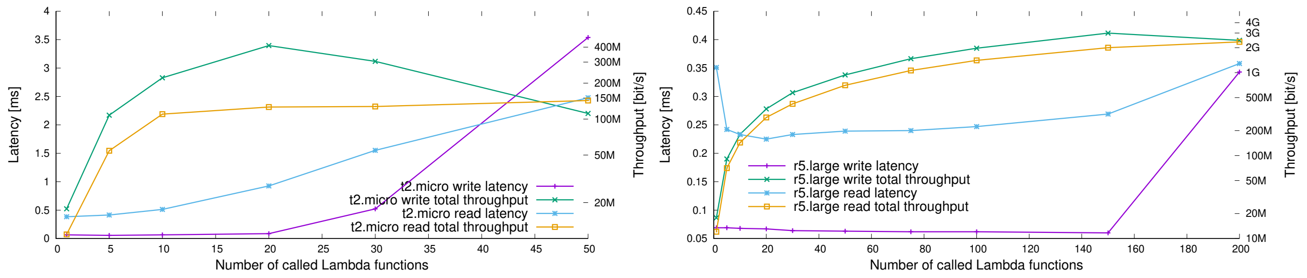


Figure 8: CDFs for selected ElastiCache scenarios.



(a) Performance of Redis.



(b) Performance of Memcached.

Figure 9: Performance measurements of different ElastiCache in-memory data stores using the t2.micro (left) and r5.large (right) flavors.

E. Data Store Throughput Tests

Following the methodology highlighted in Section III-E, we investigate throughput performance of available in-memory storage options, as depicted by Fig. 9. We note that two different vertical axes are used for latency and throughput, respectively.

a) Redis: The performance characteristics of Redis in terms of latency and throughput are shown in Fig. 9a. In this scenario, we accessed the same key during the measurement. The left part of the figure, corresponding to the smallest flavor, indicates that the total throughput limit is reached at around 10 concurrent functions while the average latency for both the read and write operations do not increase compared to accessing the storage with a single function. Using the r5.large flavor, results show significant performance improvements as displayed by the right part of the figure: we reach much lower latency and much higher total throughput. At 150 concurrent functions the latency is still lower than running 30 functions with the smaller flavor. However, it is interesting that even with 200 concurrent functions, the flavor's nominal throughput of 10 Gb/s is still not reached. The performance of our third cache node type (m4.2xlarge, not shown in the figure) was between the two previous configurations. From these observations we can conclude that increasing computation performance is only beneficial when accessing the same key with a few concurrent functions. As the number of concurrent functions increases, the flavor having better networking capabilities takes the lead in performance. We also tested access to different keys stored on a t2.micro node. Each function instance accessed a distinct key and we observed that the total throughput and latency of read operation are slightly worse compared to the single key use-case. Write performance is similar in both cases until 10 concurrent functions, and above that, the multiple keys configuration is slightly better.

b) Memcached: The single key test case of Memcached for the smaller flavor (see left side of Fig. 9b) shows that the flavor's peak performance is reached at around 20 parallel functions accessing the storage. Above that, the node is overloaded and a serious performance degradation can be identified. The performance of Memcached can also significantly improve when bigger flavors are used as it is confirmed by the plots shown in the right part of Fig. 9b. Comparison with an m4.2xlarge instance shows similar characteristics as with Redis while accessing different keys does not have a significant performance effect on Memcached. The differences stem from the internal operation of the two data stores.

V. Discussion

We argue that our measurements capture the main performance characteristics of key AWS components which could be a relevant input for developers building latency sensitive applications. In the following, we illustrate this by showing a cloud implementation of the drone control use-case discussed in Section II-A. First, as a base-line, we created a deployment of the application shown in Fig. 1 on an on-premise cloud. A webcam capable of VGA resolution recorded the movements of a Cheerson CX-10A RC drone and Python code performed position detection (with the help of OpenCV) and control while radio control was done by an Arduino device. Our experiments showed that the drone is capable of stable flight in case it receives control messages at least 30 times in a second. In practice, this means that more than one control decisions might be drawn based on the same image coming from the webcam. Position detection ran in around 27 ms while the rest of the application took around 5 ms to complete. As components were in cascade, end-to-end latency was around 33 ms which satisfied controllability requirements. These limitations are results of using the selected drone which is really sensitive to environmental changes.

We measured the execution time of each component running as Lambda function on AWS and the performance was slightly better than the local behavior. Based on our measurements, executing the code in ECS containers would not improve the performance significantly either. Since round trip time between the nearest AWS location, Frankfurt, and our premises is 14 ms, this implies an approximately 45 ms total end-to-end latency which renders perfect control impossible. By leveraging the parallelization capabilities of Lambda, however, adequate update of control instructions could still be reached. As Fig. 10 shows, we grouped control functionality into two Lambda functions. Position detection was kept separately while Manager and Controller were merged. Redis, being the quickest option for accessing small sized data with high frequency, is used to store all information. A t2.micro instance is used for storing values in 9 different keys. The Trajectory control entity is realized as an ECS container which has to provide a constantly accessible GUI. The Manager & Controller function accesses current and target drone positions as well as its state information from Redis and writes control information to a DynamoDB instance. From here, the locally running software that controls the Arduino radio reads actuator control. Both Lambda functions and the local instance are triggered every 30 ms for which we use custom triggers since AWS's built-in trigger events do not support subsecond intervals. With this setup, the drone was guaranteed to receive control messages at the right frequency. Fig. 11 shows a screen grab from a live measurement indicating 65 ms control latency as a best case. In this scenario, the overall latency comprises 14 ms of network delay, ~20 ms of processing delay caused by accessing AWS services and ~30 ms for executing the drone control code. This end-to-end latency caused the real-life drone to drift away from the desired trajectory in a matter of seconds. We investigated the performance of the cloud control application using an emulated version of the drone as well. We designed

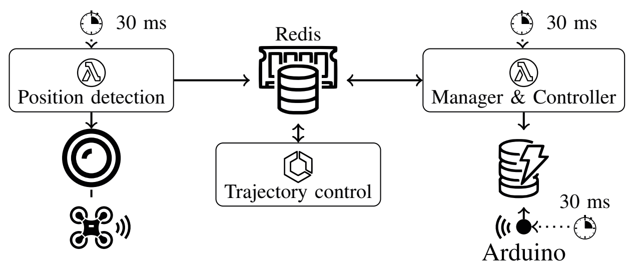


Figure 10: Drone control implementation over AWS. (Solid lines: read/write operations, dotted lines: triggers.)

the emulation to be less susceptible to environmental changes than its real-life counterpart and ran it in the Ireland AWS region. As we kept the control application in the Frankfurt region, we accomplished approximately the same network delay compared to the live scenario. Plotting target and actual coordinates showed that the emulated drone oscillated around the target coordinates, however our application managed to keep control over it, as opposed to the real-life scenario.

VI. Related Work

Although AWS Lambda is not specifically designed for latency sensitive use-cases, with appropriate software design it still can be a suitable choice. For example, such an application is demonstrated in [6] where low-latency video encoding is implemented over AWS's Lambda and EC2 services. Our performance analysis on AWS FaaS/CaaS offerings and related services could provide input for designing such applications. Performance evaluation of these services is presented in multiple works, however, not all components are investigated which could affect the overall latency and multiple options for similar functionality are generally not considered. Our work examines performance of code written in Python (which other works ignore). [9] focuses on response time of open source serverless computing frameworks while keeping a keen eye on performance during autoscaling. [8] discusses different states of warm or cold functions and their effect on response time as well as gives an analysis on provisioning, resource reservation and load balancing at AWS and Microsoft Azure. [4] offers a JavaScript solution for benchmarking execution times of FaaS offerings. It also evaluates performance and cost properties of offerings from different public providers. For benchmarking, different function types are used that target different use-cases. [10] measures throughput and latency parameters of AWS Lambda and the impact of scaling on three sample network functions: packet counter, firewall and IDS. A comparison of different invocation methods is also lacking in current research while no detailed comparison of data storage options and their performance is given either. [5] analyzes the performance of Memcached in a non-cloud environment. [3] gives an analysis about the characteristics of workloads hitting key-value stores based on Facebook's Memcached deployment. It discusses typical request sizes and composition, access patterns and hit rates that are collected in order for designing a model for generating synthetic workloads. [7] investigates the suitability of certain storage services for serverless analytics. Among

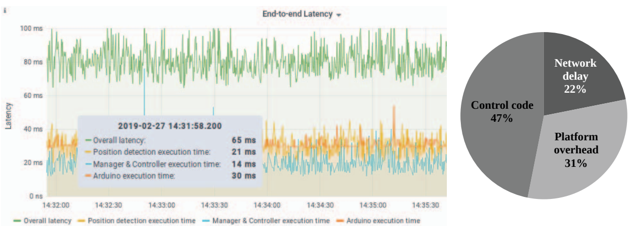


Figure 11: End-to-end latency in the drone control over AWS.

AWS services, the throughput, and read/write properties of S3 and Redis over ElastiCache are addressed. Since our work focuses only on AWS, we were able to dig deeper and provide a comprehensive analysis of AWS services.

VII. Conclusion

In this paper, we provided a comprehensive performance analysis on Amazon's public cloud platform. As we addressed latency sensitive applications, the main focus was on delay characteristics. We found that the selection of cloud services when building the application can be crucial. There are several alternatives of similar services with significantly different performance characteristics. The cloud platform itself contributes to the overall latency mainly by invocation times and external data access times. Moreover, the latency is increasing with the amount of data depending on the given technologies, and critical thresholds can easily be exceeded. In general, the variance and the jitter can be really high. For example, some measurements indicate multiple operation modes with delays centered around different values, which makes it difficult to predict application performance. These jitter related issues should be handled in the applications. To sum up, if we choose cloud services carefully at design time and the application tolerates delays up to $\sim$100 ms, then the cloud native approach can be feasible with all of its advantages.

References

[1] Amazon ElastiCache features. https://aws.amazon.com/elasticache/features/. Accessed: 2019-02-18.

[2] Invoking Lambda Functions. https://docs.aws.amazon.com/lambda/latest/dg/invoking-lambda-functions.html. Accessed: 2019-02-18.

[3] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. 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, pages 53–64, New York, NY, USA, 2012. ACM.

[4] T. Back and V. Andrikopoulos. Using a Microbenchmark to Compare Function as a Service Solutions. In *ESOCC*, 2018.

[5] V. Chidambaram and D. Ramamurthi. Performance Analysis of Memcached.

[6] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 363–376, Berkeley, CA, USA, 2017. USENIX Association.

[7] A. Klimovic, Y. Wang, C. Kozyrakis, P. Stuedi, J. Pfefferle, and A. Trivedi. Understanding Ephemeral Storage for Serverless Analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 789–794, Boston, MA, 2018. USENIX Association.

[8] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara. Serverless Computing: An Investigation of Factors Influencing Microservice Performance. *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 159–169, 2018.

[9] S. K. Mohanty, G. Premsankar, and M. D. Francesco. An evaluation of open source serverless computing frameworks. 2018.

[10] A. Singhvi, S. Banerjee, Y. Harchol, A. Akella, M. Peek, and P. Rydin. Granular Computing and Network Intensive Applications: Friends or Foes? In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, pages 157–163, New York, NY, USA, 2017. ACM.