Edge Architecture for Dynamic Data Stream Analysis and Manipulation

Orpaz Goldstein[1,2], Anant Shah[2], Derek Shiell[2], Mehrdad Arshad Rad[2], William Pressly[2], and Majid Sarrafzadeh[1]

[1] University of California Los Angeles
orpgol@cs.ucla.edu majid@cs.ucla.edu
[2] Verizon Digital Media
first.last@verizondigitalmedia.com

Abstract. The exponential growth in IoT and connected devices featuring limited computational capabilities requires the delegation of computation tasks to cloud compute platforms. Edge compute tasks largely involve sending data from an edge compute device to a central location where data is processed and returned to the edge device as a response. Since most edge network infrastructure is restricted in its ability to dynamically delegate computation while retaining context, these events are commonly limited to a predefined task that the edge function is modeled to process and respond to. Edge functions traditionally handle isolated events or periodic updates, making them ill-suited for continuous tasks on streaming data. We propose a decentralized, massively scalable architecture of modular edge compute components which dynamically defines computation channels in the network, with emphasis on the ability to efficiently process data streams from a large amount of producers and support a large amount of consumers in real time. We test this architecture on real-world tasks, involving chaining of edge functions, context retention, and machine learning models on the edge, demonstrating its viability.

1 Introduction

The recent proliferation of IoT devices has been complemented by the rapidly increasing development of serverless ecosystems, in addition to a variety of architectures for managing the complexity of pervasive IoT [1,2].

As we connect more and more internet dependent devices that require continuous, low latency connection to the network, congestion on centralized cloud servers and increasing bandwidth requirements have pushed the creation of the edge network paradigm. The edge in all its variations, moves the previously centralized services to a physically closer location to the end user. Operating on the edge of a network in this way, allows for a decentralized approach that greatly benefits its users in 4 distinct areas: Increasing bandwidth and responsiveness, high scalability, increased privacy using local policies, and outage mitigation [3].

Multi-access edge computing (MEC) is an example of an edge network architecture in active deployment stages that is gaining popularity [4]. MEC utilizes

available infrastructure used for radio access networks (RAN), and adds computation and storage capabilities to nodes that are already physically close to end users. In order to control latency and bandwidth on access to these nodes, MEC is expected to be paired with emerging 5G technologies to support traffic to nodes. While operating on the edge of the network, MEC support for computation delegation is usually passed to centralized cloud services, although to much lesser extent than current computation delegation [5]. In most cases MEC makes up the gateway access to a providers core network services that are physically further away in the hierarchy of the edge/cloud network. Effective use of MEC still requires an intermediate layer, closer than a central cloud. One possible approach then could be pairing MEC as a first layer of access to a larger edge network.

In the serverless world, the idea of functions as a service (FaaS) is rapidly becoming the preferred solution for IoT use-cases. Typically, serverless functions are of limited expressiveness and are designed to scale, preventing state information from being stored between executions [6]. Data stream related computation delegation is in turn thought of as a rigidly defined task delegation. Unlike standard FaaS usage, we are interested in defining computation paths for pipe-lining execution of edge models and functions, and provide a mechanism to orchestrate this execution and exchange of meta-data between functions and models. Data streams related tasks, such as video augmentation and analysis, might benefit from function chaining while retaining context, with multiple forking of tasks based on slightly different final product requirement of different consumers. Or alternatively, for a consumer who relies on data produced by multiple producers. Another example are context dependent models on the edge. In order to train and test machine learning models delegated to an edge network, context must be retained and potentially shared between locations.

For example: Imagine a network connected camera that uploads a live video feed to the network. One consumer wants to run a facial recognition model on the feed, and another wants to augment the feed and add bounding boxes to elements in the feed. Each of these consumers is able to define a function/model that directly subscribes to the availability of that video feed frames on the network. Once they are available, each function picks them up and computes a result that is in turn published as available on the network for the original and additional consumers to pick up.

We then wish to retain some contextual information while data is handed off from function to function, and eventually returned to a consumer. Recent work suggests the addition of ephemeral storage for short-lived edge compute tasks to achieve near real-time performance [7]. This fine-grained scalability appears to be key in developing future serverless applications that could both process multiple data streams in parallel and achieve real-time performance. Whether that refers to facial recognition on mobile devices, flying drones, or driving smart cars, support for this computation with low latency is crucial [8]. Additionally, since users of an edge network will be geographically distributed, low latency availability of an edge function should be unbound to a specific location or edge

node. Similarly, data produced in one location should be simultaneously available as input to functions and models across all edge nodes.

For example: If a user is training a facial recognition model on the edge using video feed from a mobile camera that he carries with him, that model should be available with low latency regardless of a user's physical location. If multiple users are training the same type of facial recognition model, it might make sense to share the data stream with all users globally. Conversely, if the data stream is private, we should make each of the mini models available globally and utilize what they learned to minimize training time across the board.

Figure 1 plots our data stream use-cases over a desired edge network architecture, where data produced is globally available, and any edge function/model can utilize output from other functions and continue computation while retaining context.

Our contribution in this paper is multifaceted:

i We maintain edge-level low latency and availability to physically close users, while extending the availability of produced data streams globally with low latency, without going through a centralized location.

ii We extend the definition of computation on an edge network to be more dynamic in nature. Delegated computation or usage of a function or a model that is not on a user's local node should be handed off in-network to potentially multiple locations for added efficiency, instead of reaching a centralized data center. Delegated computation should communicate meta-data back and forth to coordinate.

iii We provide an architecture where a produced data stream, or the output of a model that takes that stream as input, is available to be consumed by multiple consumers globally. Similarly, input to an edge function that depends on multiple data streams produced in various geographical locations is available instantly. Consequently, to support this kind of global availability, a modular approach to computation delegation is considered. Supporting modularity of the edge, manifested in chaining of edge functions and decentralized learning models on an edge network, requires adding context retention to the edge.

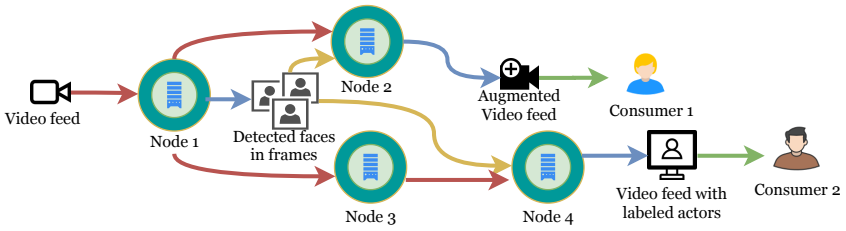

Fig. 1. Red arrows represent the original video input available to edge nodes, yellow arrows are intermediate context being fed along with data, blue arrows are outputs from edge functions/models, and green arrows represent data consumed by an end consumer. This figure shows video feed uploaded to an edge network, where it is ingested by different nodes, each potentially outputs a result that in turn could be again consumed.

2 Related work

A lot of recent work that suggests improvement to current edge architecture is centered around reducing latency and increasing efficiency. This could be done by combining the availability and low latency quality of the edge while inheriting the advantages of data center based service delivery [9], or by moving away from a centralized cloud approach to a more decentralized one [10]. On top of that the issue of Quality of Service (QoS) and understanding the benefits of offloading computation within an edge network becomes important when scaling up service to more and more IoT clients. An approach to periodically distribute incoming tasks is described in [11], showing that internally distributing tasks can result in a larger number of tasks processed. [12] extends the notion of offloading computation and computes a delay metric to find the best neighbor node in a network to offload to. Multi-access edge computing (MEC) is surveyed in [13] as a promising target for improving performance of delegated compute to an edge network, and compares different MEC concepts in terms of computation and control placement.

Running models that require state retention on an edge network could be challenging to orchestrate. In the centralized case, federated models were proposed to compute an aggregate of all model updates and broadcast them back to the sub models [14]. However this centralized approach will not work well on streaming data, or generalize to all possible state retaining applications. In the decentralized approach, some work addresses the need for internal communication and passing of information between models. [15] explores the the benefits of message passing to compute the same federated aggregation and efficiently compute a decentralized federated model. [16] discusses the treatment of data streams on an edge network for the consumption of learning models. Locality of computation offloading as well as the minimization of raw data routed to a centralized location is highlighted as necessary for overall performance of IoT supporting edge network. Our work in this paper presents a design that adheres to the same decentralized approach, focused on maximizing efficiency in handling data streams from multitude of clients.

3 Proposed Edge Architecture

3.1 Background

A natural candidate to provide the foundation of an edge network is a content delivery network (CDN). A CDN can be seen as a specialized use-case of an edge network, as it is a low latency distributed network in close physical proximity to consumers. A CDN is concerned with caching content as close as possible to end users so it could be consumed by multiple consumers with the least possible latency. Unlike the multi purpose edge network, a CDN does not provide clients with an access point into its network. A CDN does not outsource computation to users as a service or allow them to upload any code to the CDN network. To utilize a CDN as an edge network, low latency edge nodes that allow users

access into the larger network are needed, combined with support for requesting compute resources.

3.2 CDN as a platform for EdgeCompute

We propose an edge network implemented over an existing large commercial content delivery network (CDN). By leveraging an existing global network of points of presence (PoPs) that are deployed in large metro areas around the world, we get physically close to a large portion of the population on the planet. We can then construct a globally available edge presence with exceptionally low latency from outside of the network to edge nodes, as well as internally between our PoPs, from edge node to edge node.

We leverage existing CDN features when extending the network. The CDN is made to handle load balancing of traffic while taking latency into consideration. A CDN has built-in support for routing incoming traffic to the nearest PoP with the capacity to process the request efficiently. Traffic routing and management, as well as fail-overs from PoP to PoP are then taken care of by CDN logic. Since the CDN has a global presence, that translates to low latency hops globally. For an edge network user, that means that while he only maintains a connection with a local edge node, he can still benefit from a low latency global computation delegation. A CDN network has valuable security features in place, such as web application firewall (WAF) and authentication to our network. Further benefits include rate limiting of traffic, and the ability to use the CDN cache when necessary. Inherently, edge compute traffic enjoys the same benefits. Lastly, we make use of a load aware auto-scaling mechanism. On a CDN, when a piece of data becomes popular and frequently requested, it makes sense to replicate that piece of data to more cache servers so it could be served more efficiently from more servers without hurting the performance of the network. The auto-scaling mechanism is used when scaling up our edge compute tasks and as we describe later, when we augment the network with a new kind of data store.

3.3 Extending a CDN

Virtualization In order to support edge compute on our network and generalize CDN services, we allow users to upload code to be run on our network in a virtualized environment. Allowing each machine to support multiple users operating in isolation on the same hardware resources, we bound models/functions to a user-space container on a machine. The container approach for OS level virtualization of resources is highly scalable, and can be further improved by container orchestration software, automating global management and scaling of containers. Containers are fast and easy to deploy using provided packaging and deployment tools, while allowing for individualized system configuration at deployment time. Containers requires small amount of resources to maintain, and their footprint on a system is minimal. We use Docker as our container platform, and support uploading docker images containing functions or models to be run on our edge network.

Data store and context retention Unlike the common edge network implementation, the edge functions uploaded by our user do not need to integrate with an http request logic library in order to obtain data as input. Instead, we implement a distributed globally available edge key value store (EKV). Using a persistent, globally distributed data store, provides an edge network with the ability to retain context between function executions, or an online learning model to be updated from multiple nodes around the network. An EKV provides a producer with a low latency access point to upload data streams, after which the data propagates through the edge network quickly to become available globally. Equivalently, consumers are able to access data streams produced remotely, on their local edge node instantly.

Figure 2 shows our CDN based architecture; Global decentralization and low latency availability is key in a network designed for massive scale data stream input. Figure 3 shows our layering scheme and the path of a user request interacting with our network. Requests from the outer layer that is close to a user propagate internally using CDN mechanics augmented with a globally available storage system.

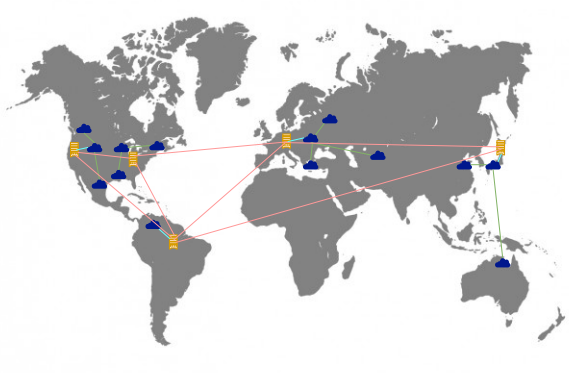

Fig. 2. Edge CDN Architecture. Red connections are strongly consistent database connections. Green connections are edge node to edge node connectivity. Blue connections are edge node to database connections.

Computation channels On the producer side, once data is produced it is pushed to a local edge instance of our EKV. Once uploaded, we wish to notify functions and models who are dependant on this data that a new piece of data is available to be consumed. In order for edge functions or models to become aware of the new data availability, we implement computation channels that are essentially named communication channels that functions or models are able to subscribe to and receive data from. In practice, to let subscribers know when to pull data from the EKV store, a user implements a publish call when a producer has finished uploading data to EKV. This allows functions and models that are subscribed to the computation channel dedicated to the data produced by a specific producer, to get data from EKV and start working. Similarly, an

implemented consumer function, model, or end user subscribes to the channel that matches the data they wish to consume.

To create computation channels, we are using the pub/sub paradigm. This approach provides us the scalability and modularity required by our implementation. Although pub/sub has some inherent rigidity as related to modifying published data, our approach allows for flexibility in the definition of EKV keys that are published via our channels. A user might publish multiple data chunks via a single key if he is not concerned about consistency, or publish a new key on every new upload if he is. Data that is augmented by a function is considered new data and is (re)published separately. Using these channels is not limited to passing EKV keys. Computation delegation across different nodes that do not require EKV data store might still use pub/sub channels to exchange meta-data between executions, pass function return values that do not require storage, and coordinate runs across different locations.

To implement computation channels, we selected the MQTT messaging protocol as our message broker. MQTT shares the IoT approach where any device is a potential client, and is flexible in using quality of service (QOS) assurances that tie nicely with a data stream approach. As our MQTT server, we use a Mosquitto broker on our edge nodes. Mosquitto is robust enough to run on our heavy duty servers supporting high volumes of messages, as well as lightweight enough for potentially running on dedicated low power edge hardware.

Figure 4 shows a demo of computational channels for data produced outside the network, and Figure 5 shows a demo of computational channels for data produced inside the network architecture. The different propagation paths of the data uploaded to the EKV store and the MQTT pub/sub calls are denoted using color arrows. This representation is meant to capture the concurrency of our network and the emphasis of global availability.

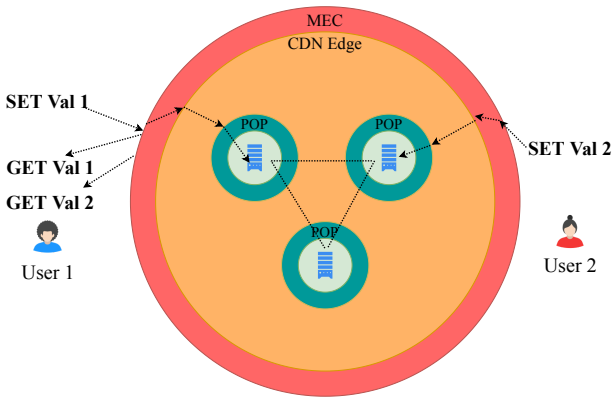

Fig. 3. Edge Architecture layers. Extending current CDN architecture using a global persistent database. MEC outer layer allows low latency edge node computation and access to larger CDN edge network. PoPs and EKV instances are interconnected. A value computed across the network is still available for local consumption.

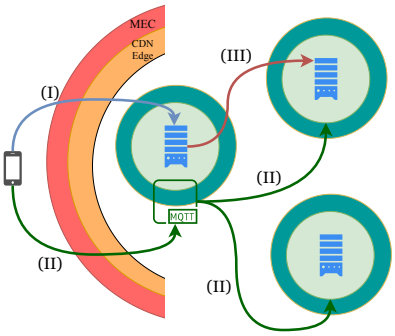

Fig. 4. Computational pathways incoming demo. (i)Smartphone uploads text data (blue arrow) and publishes availability via MQTT broker (green arrow). (ii)MQTT broker forwards the publication (green arrows) and EKV store makes data available to all edge nodes. (iii)Once the subscribed edge function receives MQTT publications, it pulls the text data (red arrow) and runs compute.

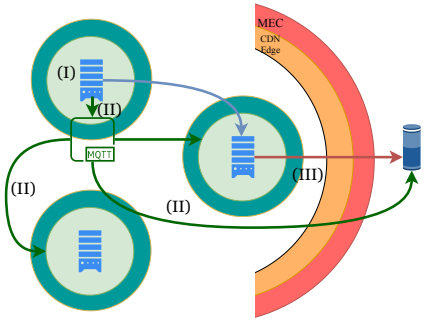

Fig. 5. Computational pathways outgoing demo. (i)An edge function finishes computation, it pushes outputs data to EKV (local so no arrows), followed by a MQTT publish call (green arrows). (ii)Finally a subscribed user receives the publication and requests the data from EKV, which retrieves it from source (blue arrow), and then allows user to pull (red arrow).

3.4 Resulting set up and real world example

The described set up allows a highly dynamic computation pipeline on the edge. The subscription to computational channels could be as hierarchically complicated as needed, using multiple layers deep of edge computation subscriptions. This allows for fine grained customization of computation that could be in turn individualized up to a per user case. Additionally, this allows for invocation of highly localized edge functions or models that are physically far away, but are on the same network and have access to the same EKV. A motivating example would be sharing a trained model without having access to the private data it was trained on, for instance: if we want to train a model on the edge, we might benefit from utilizing models on EKV that did something similar in different geo-locations. This can be seen as a debiasing stage that is private and edge contained. Federating models prevents bias that arises from locally collected training data, and sharing models on the edge network instead of data keeps that data private. Another advantage of our network is the ability to retain context and make it available globally. Since we allow subscribing to computation results of another edge function or model, we sometimes need to maintain the proper context in addition to the output on EKV.

For example: Say we are feeding a video to the edge, and a function is subscribed to detect faces in frames of the video feed. The output of that function is the coordinates of a bounding box for a face in the frames. Now, if a function is subscribed to the results of that face detection function and is planning to use the face detection results and continue to augment the faces on these frames,

it will need both the coordinates produced by the face detection function in addition to the original frames on EKV. To support such a use-case we need to understand what a subscription to a computational channel depends on. In this case, that subscription to the output of face detection is dependant on the original frames being available in EKV. We solve this by having the augmenting function subscribe to two computational channels and starting work when both a frame and its corresponding coordinates are available. Lastly, all computation is done on the edge whether local or remote. There is no delegation to a centralized cloud. Instead, any non local edge node might potentially participate in computation if such delegation is needed.

Figure 6 and 7 show a demo of a text-to-speech task execution on our architecture. The different propagation paths of the data uploaded to the EKV store and the MQTT pub/sub calls are denoted using color arrows. The latency observed on MQTT publication and data transfers from outside the network to a PoP with an EKV instance, and from EKV to another PoP that had our Text-to-Speech function is denoted on the plot. This representation captures our real world experimentation with our architecture and the latency observed.

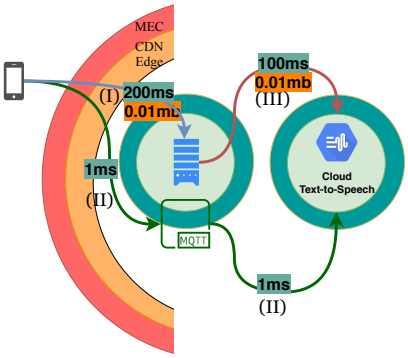

Fig. 6. Text2speech incoming. (i)Smartphone uploads text data (blue arrow) and publishes availability via MQTT broker (green arrow). (ii)MQTT broker forwards the publication (green arrow) and EKV store makes data available to all edge nodes. (iii)Once the subscribed edge function receives MQTT publications, it pulls the text data (red arrow) and runs compute.

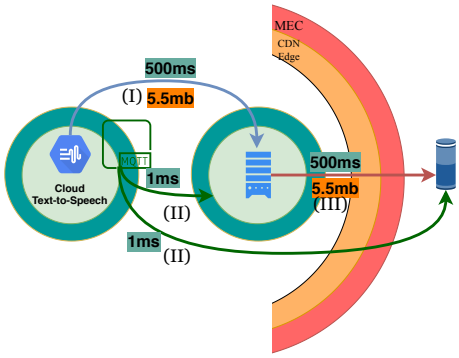

Fig. 7. Text2speech outgoing. (i)Text-to-speech model finishes run, it pushes outputs data to EKV (local so no arrows), followed by a MQTT publish call (green arrows). (ii)Finally a subscribed user receives the publication and requests the data from EKV, which retrieves it from source (blue arrow), and then allows user to pull (red arrow).

4 Experiments and Measurements

Our architecture is built to support large amounts of data stream traffic and computation paths within the network. We show in our experiments the advan-

tages of our network in a few key edge related tasks. We show how close we can get to real time delivery of results from edge functions and models working on analyzing data streams. We evaluate the efficiency of chaining different functions while retaining meta-data between executions. And we evaluate how close we can get to real time generation of data from a machine learning model, based on ques from a user outside our network. The tasks are as follows:

1. Run an emotion detection (ED) model as an edge function on recorded voice samples from an IoT device and show detected emotion in real time.
2. Run a text to speech (T2S) function on the edge that accepts text from a user and outputs generated human voices that will then be consumed by a second IoT device.
3. Pipeline 3 image related machine learning model that will accept as input a video stream and output an augmented version of it (DF).

For each of these we report all metrics relating to latency and connectivity throughout the path of execution. All experiments were run on our 32 core Intel(R) Xeon(R) Gold 6140 CPU @ 2.30GHz, with the last two tasks utilizing a single 8GB NVIDIA Tesla M60 GPU. The edge node we are using in these experiments is located in close physical proximity (Los Angeles) and within 5ms to the client. The EKV instance is located remotely (Chicago) and within 7ms to client. We show a few machine learning use-cases that take advantage of our proposed edge architecture. Models or functions running on our edge can seamlessly integrate into a user's edge function chain.

Table 1 shows a comparisons between the different experiments and their key points we are interested in evaluating on our edge architecture.

Table 1. Experiment Comparison

	ED	T2S	DF
Pub/Sub	✓	✓	✓
EKV Data Store	×	✓	✓
Function chaining	×	×	✓
Small files incoming	✓	✓	×
Small files outgoing	✓	✓	×
Large files incoming	×	×	✓
Large files outgoing	×	✓	✓
Model is target specific	×	×	✓

4.1 Emotion detection from voice samples

With the expanding array of smart speakers consumers interact with, voice analysis becomes a common task for extracting commands, as well as features from voice that are unrelated to the spoken text. Here, we evaluate here the usage of a machine learning model trained to classify positive and negative emotions on the edge and return a response in real time. This task shows low latency availability of our emotion detection model, and utilization of computation channels to exchange small amounts of data and coordinate execution.

Using a database of labeled actor voices emulating emotions, we pre-train our model and package it into a Docker container deployed on the edge. The dataset we used is described in [17]. We create two MQTT channels. One for incoming data and one for outgoing classification. We then create and subscribe an edge function that listens to incoming publications and branch an instance of the emotion detection model for each incoming request. On the client side, we create a web page for recording voice samples using JavaScript, and use paho-mqtt in the browser to subscribe to the channels for this process. The voice samples are recorded trough the browser, serialized and sent through MQTT as the payload of the MQTT publish call. The edge function receives the serialized file, passes it to the emotion detection model, and publishes the classification results via MQTT. The browser then receives the publication and displays an emoticon on screen [3]. The voice sample size depends on the recording length, but we have found that a file of about 100kb could be published via MQTT and received by the listener function in less than 3ms. After the function loads the pre-trained model and passes the data, the model takes about 4.5 seconds to run using only the CPUs on the edge node. The model output is then published back received by the client after another 2ms and finally the client displays an emoticon after about 4.5 seconds from the end of the recording process.

As a comparison, we consider a standard cloud architecture, with serverless functions that accept data via HTTP requests. The ping to an average cloud instance takes an average of 224ms [4], and response time per message using HTTP is 200ms higher than that of MQTT when connection is reused [5]. Including the time it will take to move the data makes our task potentially 1-3 seconds longer on public cloud. A significant user impact.

4.2 Text to speech

Offloading the resource intense process of data generation using a relatively small amount of data is another area where edge networks shine. Asking an edge node to generate images for us using a description, or generated human voice from raw text are only two examples, out of which we will examine the latter here. This task shows a producer and consumer that are operating independently in different locations using different edge nodes for EKV and edge function. The edge nodes used are in Chicago and California and are 1ms apart when using ping.

Using a model based on Deep Voice 3 [18], we create an edge function that is able to receive a blob of raw text and parameters indicating what kind of speaker the model should generate, and outputs a recording of human voice that speaks the text that was received. We create two MQTT channels, one for publishing blobs of text, and the other for publishing human voices. Our edge

[3] A video showing the emotion detection task can be seen here

[4] Measured using 'curl' to 30 public cloud instances from different companies and averaged

[5] Detailed comparison can be seen here

function subscribes to the text channel, and waits for a publication that a new blob of text is available on EKV along with the parameters of what voice should be generated. An instance of our function starts for each blob/speaker pair. We create a producer of text that pushes text to EKV and publishes on the text channel, and a separate consumer that subscribes to both channels and consumes both the text and the corresponding human voice that our edge function outputs. This simulates a situation where the producer is not the final destination for the processing result on the produced data. We run the producer and consumer in separate locations, and time the task of a producer pushing a blob of text and asks for 10 different human voices generated for each blob. The producer pushes approximately 200 bytes of text to EKV and receives confirmation within 200ms, once the data is on EKV the producer publishes via MQTT that a new blob is ready for consumption. The edge function pulls the text from EKV and starts 10 instances of text to speech translation after about 500ms. The consumer receives the publication and prints out the text after 250ms of process start. The edge text to speech function outputs 10 .WAV files weighing a total of 5.8Mb after working for 4 seconds. It then uploads them to EKV simultaneously and receives a final confirmation from EKV after a total of 5.5 seconds from process start. The function then publishes to MQTT the availability of results. Once the consumer receives the publication of new available data, it pulls the data from EKV and saves them locally after about 6.3 seconds from process start [6].

To compare, we look at the average latency between nodes of popular cloud services [7]. In addition to the 224ms average ping time from client to cloud service and 200ms longer response time per message, average latency between public cloud nodes is approximately 160ms. Depending on implementation of storage and upload/download of data, our task will take at best seconds longer on the average public cloud.

4.3 Video stream manipulation

Leveraging the edge as a live video manipulation tool opens the doors for many interesting use-cases such as dynamically augmented video streams. Combining that with machine learning models such as Deepfake, lets us imagine a future where we consume personally tailored video streams, replacing actors in a movie we are watching on the fly. We will examine how we can use our architecture, and create a pipeline of functions to create a augmented version of a video stream as close to real time as possible. This task evaluates chaining of functions and models to augment a video feed live. Producer and consumer are operating independently in different locations using different edge nodes for EKV and edge function. The edge nodes used are in Chicago and California and are 1ms apart when using ping.

We define 4 computation channels. We have an edge function and two edge models chained, each subscribed to the output of the previous functions, and an

[6] A video showing the text to speech task can be seen here

[7] latency average was computed based on information in: https://www.cloudping.co/

extra channel publishing the availability of the original frames on EKV that all are subscribed to, thus creating a pipeline of computation for the video frames to go through. We also define a helper edge function that extracts individual frames from a video clip using FFMPEG library [8].

(i) Face detection First, we have a function based on OpenCV face detection that accepts video frames as input and outputs location of faces in frames. Once a frame is passed to the function it is passed to OpenCV where frames are rotated and scaled multiple times as the OpenCV detector function scans for faces. Coordinates for detected faces are saved on EKV.

(ii) Face classification Coordinates for faces that are identified by the previous function as well as the original frames are ingested by a model based on VGG face classification [19] pre-trained for face classification using 2.6M images from 2622 identities. Our model is used to identify a specific face of interest that we wish to augment. It accepts a frame with a face and a reference image, and outputs whether the face in the frame matches our person of interest back to EKV.

(iii) Face augmentation Frames that were marked by the classification model are then picked up by our Deepfake model that is based on work of [20]. The model uses the stored coordinates for each frame to extract a cropped face to convert. The model then performs conversion of the frame, reconstructed to fake the source face in the original frame into the desired target face, and output the augmented frames back to EKV.

Data used for training of our model has been scraped from YouTube videos of the original source face and the target face, and in total used to create around 5000 images of each. The model is trained by us for one week using a single NVIDIA Tesla M60 GPU before compiled as an edge function.

(iv) Consumer A consumer that is subscribed to the original stream in addition to the output of the deepfake function, is able to pick up the video feed with augmented frames from EKV and view the feed locally.

Timing The producer streams 20 second chunks of video to EKV weighing an average of 1.9Mb. A single chunk takes about 300ms to upload to EKV and receive confirmation for. The producer then publishes via MQTT that the chunk is ready for consumption. Face detection then picks up the chunk and starts the process of detecting faces. Coordinates of each face detected is immediately pushed to EKV and published as available. The first video chunk takes about 36 seconds to process due to model loading, and each following chunk will be processed within 300ms of producer pushing to EKV. Face classification receives

[8] A video showing the augmentation task can be seen here

publication and within less then 20ms classifies the face pushes result to EKV and publishes availability. The face augmentation phase then has received publications from all channels it is subscribed to and starts working on changing the faces on a chunk of video. Converting the entire 20 second chunk of video takes 18 seconds. The Deepfake model than pushes changed frame back to EKV and publishes availability within a few milliseconds. Lastly, the consumer receives the publication of availability of frames. Once there is a 20 second chunk of frames available, it uses the helper function to convert them back into a video file, downloads and plays them. From publication of faked frames, it takes the consumer approximately 500ms to convert and download the 20 second chunk of video. Overall for the first chunk, it takes about 55 seconds for a chunk of video to be augmented and viewed on the consumer end. After the first chunk it will take under 20 seconds for the entire pipeline to finish working on a 20 second chunk. We then can keep our augmented video feed about 1 minute behind live video.

Comparing to the latency on a public cloud service, In addition to the 224ms average ping time from client to cloud service and 200ms longer response time per message, and an average of 160ms latency between distant nodes, we add the accumulating latency of making the intermediate results of each function globally available for consumption. Assuming data passes via http/https, our task could not be augmented fast enough to be viewed in pseudo-live time.

5 Conclusion

In this paper we have examined the recent development in edge network design, as well as the role of the edge network for the near future connectivity requirements, and have proposed an architecture to close that gap. By building our edge network on top of an existing CDN and extending it, we have constructed a massively scalable edge network. We have demonstrated the benefits of having both computational paths that can operate as meta-data exchange channels for coordination or limited size message passing. And the benefit of having a globally connected data store for context retention and intermediate data storage where learning models can read and write to and keep a global low latency availability. Demonstrating the above, we have provided 3 real world machine learning tasks making use of computation paths and context and data retention.

References

1. Lucero, S., et al.: Iot platforms: enabling the internet of things. White paper (2016)
2. Shi, W., Dustdar, S.: The promise of edge computing. Computer **49**(5) (2016) 78–81
3. Satyanarayanan, M.: The emergence of edge computing. Computer **50**(1) (2017) 30–39
4. multi-access-edge computing: https://www.etsi.org/technologies/multi-access-edge-computing

5. Taleb, T., Samdanis, K., Mada, B., Flinck, H., Dutta, S., Sabella, D.: On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration. IEEE Communications Surveys & Tutorials **19**(3) (2017) 1657–1681
6. Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., et al.: Serverless computing: Current trends and open problems. In: Research Advances in Cloud Computing. Springer (2017) 1–20
7. Klimovic, A., Wang, Y., Stuedi, P., Trivedi, A., Pfefferle, J., Kozyrakis, C.: Pocket: Elastic ephemeral storage for serverless analytics. In: 13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18). (2018) 427–444
8. Shi, W., Cao, J., Zhang, Q., Li, Y., Xu, L.: Edge computing: Vision and challenges. IEEE Internet of Things Journal **3**(5) (2016) 637–646
9. Chang, H., Hari, A., Mukherjee, S., Lakshman, T.: Bringing the cloud to the edge. In: 2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), IEEE (2014) 346–351
10. Garcia Lopez, P., Montresor, A., Epema, D., Datta, A., Higashino, T., Iamnitchi, A., Barcellos, M., Felber, P., Riviere, E.: Edge-centric computing: Vision and challenges (2015)
11. Song, Y., Yau, S.S., Yu, R., Zhang, X., Xue, G.: An approach to qos-based task distribution in edge computing networks for iot applications. In: 2017 IEEE international conference on edge computing (EDGE), IEEE (2017) 32–39
12. Yousefpour, A., Ishigaki, G., Jue, J.P.: Fog computing: Towards minimizing delay in the internet of things. In: 2017 IEEE international conference on edge computing (EDGE), IEEE (2017) 17–24
13. Mach, P., Becvar, Z.: Mobile edge computing: A survey on architecture and computation offloading. IEEE Communications Surveys & Tutorials **19**(3) (2017) 1628–1656
14. Konečnỳ, J., McMahan, H.B., Yu, F.X., Richtárik, P., Suresh, A.T., Bacon, D.: Federated learning: Strategies for improving communication efficiency. arXiv preprint arXiv:1610.05492 (2016)
15. Wang, S., Tuor, T., Salonidis, T., Leung, K.K., Makaya, C., He, T., Chan, K.: Adaptive federated learning in resource constrained edge computing systems. IEEE Journal on Selected Areas in Communications **37**(6) (2019) 1205–1221
16. Mohammadi, M., Al-Fuqaha, A., Sorour, S., Guizani, M.: Deep learning for iot big data and streaming analytics: A survey. IEEE Communications Surveys & Tutorials **20**(4) (2018) 2923–2960
17. Livingstone, S.R., Russo, F.A.: The ryerson audio-visual database of emotional speech and song (ravdess): A dynamic, multimodal set of facial and vocal expressions in north american english. PloS one **13**(5) (2018) e0196391
18. Ping, W., Peng, K., Gibiansky, A., Arik, S.O., Kannan, A., Narang, S., Raiman, J., Miller, J.: Deep voice 3: Scaling text-to-speech with convolutional sequence learning. arXiv preprint arXiv:1710.07654 (2017)
19. Parkhi, O.M., Vedaldi, A., Zisserman, A., et al.: Deep face recognition. In: bmvc. Volume 1. (2015) 6
20. Pu, Y., Gan, Z., Henao, R., Yuan, X., Li, C., Stevens, A., Carin, L.: Variational autoencoder for deep learning of images, labels and captions. In: Advances in neural information processing systems. (2016) 2352–2360
21. Wang, C.: Http vs. mqtt: A tale of two iot protocols (2018)

A Comparison with different architectures

In addition to the benefits accrued by our overarching edge architecture, there is room to break down individual components and compare them to other possible design choices. MQTT is one protocol chosen from the few emerging protocols of choice for the IoT world. While we evaluated both MQTT and CoAP and found both to be comparable, we chose MQTT for our pub/sub protocol as it had better library availability and broker seleciton. We compare our choice of MQTT with an HTTP based singnaling mechanism to support our architecture. In our architecture, we make use of MQTT as a signaling channel between subscribed clients waiting on streams of data, and between edge nodes coordinating execution of models on data. The key observation here is that our MQTT connection are seldom closed, and in most cases reused many times between the time they are established and close. The comparison made in [21] clearly shows the benefits of utilizing an open MQTT connection with exponential benefits over the same use case implemented using HTTP. Similarly to [21], we investigate the difference between 1, 10, and 100 messages each weighing 10 bytes, transmitted over MQTT and HTTP, over 10 trials. This simulates transferring simple instructions and EKV data locations in our computation channels. For MQTT we connect once and reuse the same connection to communicate all subsequent messages. For HTTP we use POST requests. All communication was evaluated between an edge node and a local client, emulating a real world scenario. Figure 8 and 9 show the log scale results for speed in ms, as observed in our test. Since HTTP grows as a factor of messages passed we see the benefit of opening a single MQTT connection to be used over multiple messages.

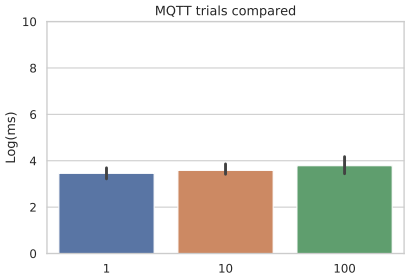

Fig. 8. MQTT speed per number of requests compared at log scale of ms.

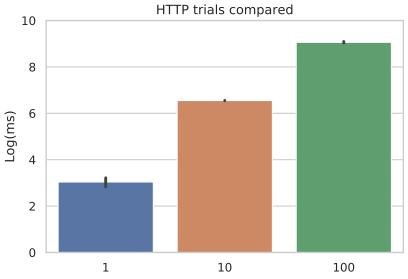

Fig. 9. HTTP speed per number of requests compared at log scale of ms.

Another aspect worthy of comparison is the speed gain of using our architecture as compared to the same job implemented as a FAAS workflow, where results must be returned back to a user before the next function in a pipeline is started. We compare a simple numpy matrix multiplication task, called via our MQTT computation channels 1,10 and 100 times, where results are pushed to a

MinIO storage instance. This is compared to the case where a function runs and returns a result directly to a client. In the case where we run our function more than 1 time, we compute the next result based on the previous functions result. In the FAAS like use case, the client sends back the result to the function, and in our architecture, the previous result is picked up from our MinIO instance. Figures 10 and 11 show the comparison between the two approaches. It can be seen that the impact on sending the little amount of data we use back and forth using HTTP POST requests, essentially does not change the POST requests time for execution. While the time increases using MQTT computation channels and ephemeral storage, where an extra call to the MinIO server is needed. However, even with this increase, it can be seen that as the amount of concurrent requests grow, the penalty incurred by POST requests is far more inhibiting then the extra hop to MinIO. As we have previously shown in our experiments, MQTT can be used for small scale data and speed up computation even more in cased where not much data is moved in the network.

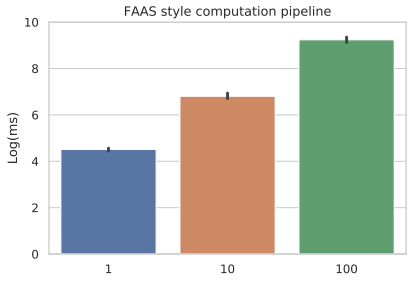

Fig. 10. Speed of calling our function via HTTP POST requests, and sending back the result for all cases where the function was called more than once. Compared at log scale of ms.

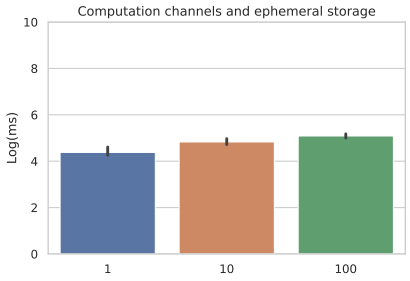

Fig. 11. Speed of calling our function via an MQTT computation channel, send result to an ephemeral storage, and compute results based on previous function run for all cases where we call the function more than once. Compared at log scale of ms.